Rapport de stage : Analyse sémantique des tableaux, des structures et des pointeurs

LOSSING Nelson
Encadré par IRIGOIN François et HERMANT Olivier
Centre de Recherche en Informatique, MINES ParisTech

22 août 2013

Le contexte général

Mon stage s'est déroulé au CRI [1] de MINES ParisTech. L'objectif de celui-ci était de réaliser des extensions de l'analyse sémantique aux tableaux, structures et pointeurs du langage `C` au sein du compilateur *PIPS* [PIP] [2]. Ces extensions ont pour but d'améliorer les connaissances sur un programme `C` en vue d'une validation ou d'une optimisation de celui-ci. Plusieurs projets de recherche sur des compilateurs source-à-source existent [CET, Loo, OSC, PoC, ROS] [3]. L'analyseur statique *Astrée* [Ast] permet également d'analyser les pointeurs du code `C` [Min06].

Le problème étudié

Cette analyse sémantique nécessite une analyse préalable de pointeurs. Il existe principalement deux types d'analyse de pointeurs : l'analyse *points-to*, où un objet p pointe vers un objet i ; et l'analyse d'alias, où deux objets p et q pointent vers la même zone mémoire. Le choix a été fait, dans *PIPS*, de développer une analyse *points-to* pour l'analyse de pointeurs car elle est directement utilisable pour effectuer des optimisations de code.

De nombreuses analyses de pointeurs ont déjà été conçues et implémentées par le passé. [HP00] et [Hin01] recensent la plupart des analyses réalisées durant les années 80 à 2000. Le travail entrepris au sein de *PIPS* ne prétend pas être la meilleure solution possible. Mais, on souhaite que ce soit la meilleure analyse possible pour un problème donné. En effet, selon la prédiction de Landi et Ryder [LR04], les nouvelles analyses ne seront pas optimales pour toutes les applications possibles, mais au contraire pour des types d'analyse particulière :

> « *We prougeict that the future will not see a* best Pointer May-alias algorithm *whose results are suitable for* any *application, but rather algorithms designed* to optimize the tradeoffs to best meet the requirements of some particular application. »

Ainsi, la question traitée par *PIPS* est l'optimisation de code `C` par génération de nouveau code `C` tout en garantissant que ce dernier reste facilement compréhensible par un être humain. Les programmes ciblés sont principalement des programmes scientifiques, en particulier des applications de traitement du signal ou d'images.

1. Centre de Recherche en Informatique, département Mathématique et Système, `http://cri.ensmp.fr/`
2. *PIPS* est un *framework* de compilation de code source-à-source développé au CRI. Il permet d'analyser et de transformer des programmes en `C` et Fortran, `http://www.pips4u.org/`
3. `http://pips4u.org/related_projects.html`

La contribution proposée

L'analyse sémantique réalisée étend l'analyse *points-to* implémentée par Mensi [Men13], à laquelle il manquait une formalisation. Dans un premier temps, j'ai ainsi décidé d'étudier sa thèse et d'effectuer la formalisation concernant l'analyse *points-to*. Contrairement à Mensi, j'ai donné une interprétation au référencement (*Sec.* 1.3.3). De plus, pour la formalisation, j'ai simplifié et corrigé les treillis (*Sec.* 2.3). Je me suis néanmoins arrêté à la partie intra-procédurale, faute de temps, ce qui permet déjà d'analyser de nombreux programmes. Conjointement à cette formalisation, j'ai étendu l'analyse sémantique. Enfin, j'ai réalisé l'implémentation dans *PIPS* afin de tester le travail effectué.

J'ai également étudié l'analyse réalisée par Miné [Min06] pour *Astrée*. Son analyse, contrairement à la nôtre, porte sur des codes pour applications embarquées dont il faut vérifier la sûreté. Ainsi, il est plus concerné par l'analyse des unions et ne traite pas l'allocation dynamique. De ce fait, il utilise une représentation au niveau des octets. En opposition, nous cherchons particulièrement à traiter l'allocation dynamique, et l'on ne traite ni le type union ni le *cast* de type. Notre représentation est fondée sur une normalisation des variables en chemins d'accès (*Sec.* 1.2.1 et *Sec.* 1.3.1).

Les arguments en faveur de sa validité

Notre analyse est ciblée pour les applications que l'on souhaite traiter avec *PIPS*. Ainsi, elle a été conçue pour répondre au mieux à l'analyse et l'optimisation d'applications pour le traitement du signal et d'images.

La validité de la solution apportée suppose que l'analyse sémantique pré-existante et l'analyse *points-to* existante soient correctes. Ainsi, l'extension de l'analyse sémantique à l'aide de l'information *points-to* sera également valide.

On a pu le vérifier grâce à l'implémentation et aux expérimentations faites au sein de *PIPS*. Les résultats expérimentaux sont présentés en annexes.

Le bilan et les perspectives

Dû à l'architecture et au fonctionnement de *PIPS* des problèmes de performances et de mises à l'échelle peuvent apparaître. En effet, *PIPS* a sa propre représentation interne afin de modéliser à la fois des codes `C` et des codes Fortran. De plus, *PIPS* se veut très modulaire en possédant plusieurs possibilités d'analyse, de traitement et de transformation de code. Cette modularité permet de cibler le souhait de l'utilisateur, mais a un coût non négligeable.

Malgré ces problèmes, notre extension permet d'élargir l'ensemble des applications pouvant être modélisées et traitées automatiquement ou semi-automatiquement. Deux extensions orthogonales ont été implémentées. La première concerne l'utilisation de l'information *points-to* pour raffiner l'analyse sémantique sur les valeurs scalaires à plusieurs niveaux de précision. La seconde est la réalisation d'une analyse sur les variables de type pointeur.

Des améliorations restent tout de même à réaliser comme la possibilité de traiter le *cast* ou le type union. De même, l'analyse inter-procédurale n'a pu être réalisée faute de temps, néanmoins son implémentation est commencée. De plus, un raffinement de l'analyse *points-to* à partir de notre analyse peut être envisager. Enfin, des améliorations mineures comme le rendu visuel de l'analyse ou le traitement de certains cas spécifique dû à la représentation de *PIPS* restent à faire.

Le plan

Le langage C étant très complexe, on se restreindra à l'étude de sous-langages de celui-ci. De plus, pour la formalisation, j'essayerai autant que faire se peut de me rapprocher de la sémantique dénotationnelle de Gordon [Gor79], et indiquerai les équivalences avec celle-ci si possible. De même, cette présentation de la sémantique est inspirée de [Min06] et de [Men13].

Ainsi, j'adopterai une approche incrémentale, en définissant successivement trois sous-langages $\mathcal{L}_0$, $\mathcal{L}_1$ et $\mathcal{L}_2$. En *Sec.* 1, $\mathcal{L}_0$ permet uniquement d'exécuter un code purement séquentiel. Il sert de base pour notre analyse et donne notamment la sémantique pour les expressions. Puis, en *Sec.* 2, $\mathcal{L}_1$ étend $\mathcal{L}_0$ avec les conditions pour introduire les tests conditionnels et les boucles. Enfin, en *Sec.* 3, on continue à enrichir le langage étudié avec le traitement de l'allocation dynamique qui est un atout fort du langage C, mais qui est également source de problème.

Dans un second temps, je parlerai de l'implémentation réalisée, *Sec.* 4, qui permet de raffiner l'analyse existante à l'aide de l'analyse *points-to*, ou d'analyser les pointeurs eux-même.

1 Le langage $\mathcal{L}_0$: le langage de base

1.1 La syntaxe de $\mathcal{L}_0$

La sous-syntaxe $\mathcal{L}_0$, *Fig.* 1.1, de la syntaxe C permet d'effectuer des séquences d'affectations. Elle permet également d'effectuer les principales opérations sur les pointeurs, à savoir la prise d'adresse, ou référencement, ($\&$) et le déréférencement ($*$). L'opération d'accès au membre d'une structure $\langle lhs \rangle \rightarrow \langle id \rangle$ sera normalisé par $* \langle lhs \rangle \,.\, \langle id \rangle$.

$$
\begin{array}{lcll}
\langle \textit{int-type} \rangle & ::= & (\text{unsigned} \mid \text{signed})\ (\text{char} \mid \text{short} \mid \text{int} \mid \text{long}) & \\
\langle \textit{float-type} \rangle & ::= & \text{float} \mid \text{double} & \\
\langle \textit{scalar-type} \rangle & ::= & \langle \textit{int-type} \rangle \mid \langle \textit{float-type} \rangle & \\
\langle \textit{op-type} \rangle & ::= & \langle \textit{scalar-type} \rangle \mid \text{pointer}(\langle \textit{type} \rangle) & \\
\langle \textit{type} \rangle & ::= & \langle \textit{op-type} \rangle & \\
 & \mid & \langle \textit{type} \rangle[n] & n \in \mathbb{N} \setminus \{0\} \\
 & \mid & \text{struct}\{(\langle \textit{id} \rangle : \langle \textit{type} \rangle)^*\} & {}^*\ \text{postfixé représente une liste} \\
\langle \textit{unary-op} \rangle & ::= & - & \\
\langle \textit{binary-op} \rangle & ::= & + \mid - \mid * \mid / \mid \% & \\
\langle \textit{relational-op} \rangle & ::= & == \mid \neq \mid \leq \mid \geq \mid < \mid > & \\
\langle \textit{index} \rangle & ::= & \langle \textit{id} \rangle \mid n & n \in \mathbb{N} \\
\langle \textit{lhs} \rangle & ::= & \langle \textit{id} \rangle \mid \langle \textit{lhs} \rangle.\langle \textit{id} \rangle \mid \langle \textit{lhs} \rangle[\langle \textit{index} \rangle] \mid * \langle \textit{lhs} \rangle & \\
\langle \textit{variable} \rangle & ::= & \langle \textit{lhs} \rangle \mid \&\langle \textit{lhs} \rangle & \\
\langle \textit{expression} \rangle & ::= & \langle \textit{variable} \rangle \mid \langle \textit{cte} \rangle \mid \langle \textit{unary-op} \rangle\ \langle \textit{expression} \rangle \mid \langle \textit{expression} \rangle\ \langle \textit{binary-op} \rangle\ \langle \textit{expression} \rangle & \\
\langle \textit{statement} \rangle & ::= & \langle \textit{lhs} \rangle \leftarrow \langle \textit{expression} \rangle \mid \langle \textit{statement} \rangle; \langle \textit{statement} \rangle & \\
\langle \omega \rangle & \in & \Omega = \{\varnothing, \omega\} & \text{Domaine des erreurs, } \varnothing \text{ absence d'erreurs, } \omega \text{ présence d'erreur} \\
\langle \textit{id} \rangle & \in & Id = Id_{\texttt{STATIC}} + Id_{\texttt{FORMAL}} & \text{Domaine des identifiants} \\
\langle \textit{cte} \rangle & \in & \mathbb{N} \mid \mathbb{R} \mid \texttt{NULL} & \text{Domaine des constantes}
\end{array}
$$

Figure 1.1: Syntaxe de $\mathcal{L}_0$

1.2 La sémantique concrète du langage $\mathcal{L}_0$

1.2.1 L'ensemble des chemins d'accès non constant $\mathcal{NCP}$ du langage $\mathcal{L}_0$

La représentation de code à trois-adresses ([ALSU06, Section 2.8.4]) n'est pas utilisée. La raison est que l'analyseur utilisé, *PIPS*, est un compilateur source-à-source. Ainsi, on souhaite rester le plus proche possible du code originel et éviter autant que faire se peut toute normalisation. À la place, on représente les expressions par des chemins d'accès non constant $\mathcal{NCP}$. Un chemin d'accès non constant permet d'accéder à une cellule mémoire typée. Cette cellule mémoire peut aussi bien être une cellule mémoire pour une variable que pour une adresse. De plus, on considère que plusieurs chemins d'accès non constant peuvent accéder à la même cellule [4]. L'ensemble de ces chemins est défini comme suit :

$$ncp \in LOC = \mathcal{NCP} = (Name \times V_{ref} \times Type) + \{\omega\} \tag{1.1}$$

où

$Name$ représente une variable :

$$Name = Id \cup \{\texttt{NULL}\} \cup \{undefined\}$$

Id est le domaine des identificateurs, *undefined* [5] est la valeur par défaut d'un pointeur et `NULL` est la valeur pour un pointeur initialisé mais ne pointant vers rien ;

V_{ref} représente les indices des références de l'emplacement mémoire que l'on considère, c'est donc une séquence d'entiers. Il est utilisé pour les accès aux pointeurs (0), aux tableaux (indice du tableau) et aux structures (indice du champ de la structure) :

$$V_{ref} = \mathbb{N}^*$$

L'opérateur « . » permet de réaliser la concaténation de séquences ;

$Type$ représente le type de la variable, avec *type_unknown* en l'absence de celui-ci :

$$Type = \langle type \rangle \cup \{type\_unknown\} \cup \{overloaded\}$$

$\langle type \rangle$ est l'ensemble des types du langage considéré, *type_unknown* pour un type inconnu, *overloaded* pour les constantes *undefined* et `NULL`, *overloaded* peut représenter n'importe quel type de l'ensemble $\langle type \rangle$.

Le code 1 montre comment des éléments du langage $\mathcal{L}_0$ sont traduits dans l'ensemble $\mathcal{NCP}$.

```
struct list {int val; struct list * next;};
int i, t[10], *p;
struct list l, *pl;
i               //<i, (), int>                              représente une variable
t[1]            //<t, (1), int>                             représente une variable
*p              //<p, (0), int>                             représente une variable
l.next          //<l, (2), pointer(struct list)>            représente une adresse
*pl.next        //<pl, (0; 2), pointer(struct list)>        représente une adresse
pl->next        //<pl, (0; 2), pointer(struct list)>        représente une adresse
pl->next->next  //<pl, (0; 2; 0; 2), pointer(struct list)>  représente une adresse
undefined       //<undefined, (), overloaded>
NULL            //<NULL, (), overloaded>
```

CODE 1 – Exemple de traduction dans $\mathcal{NCP}$

Les chemins d'accès non constant uniques pour les constantes spéciales *undefined* (1.2) et `NULL` (1.3) sont définis ci-dessous :

$$ncp_{undefined} = \langle undefined, (), overloaded \rangle \tag{1.2}$$

$$ncp_{\texttt{NULL}} = \langle \texttt{NULL}, (), overloaded \rangle \tag{1.3}$$

4. Cette hypothèse forte permettra de simplifier la sémantique concrète de notre langage. Mais notre sémantique concrète ne présente pas comment cette résolution vers la cellule mémoire est réalisée.

5. La variable *undefined* est définie par la norme `C` sous le nom de *indeterminate*, l'adjectif *undefined* étant normalement réservé au comportement du programme. Au niveau de l'implémentation cette valeur est désignée par *undefined*, ainsi par abus, on utilisera *undefined* au lieu de *indeterminate* [ISO07].

De plus, on considère que l'on a passé un *type-checker*. Ainsi, les identifiants et les opérateurs sont typés et la fonction *typeof* permet de récupérer leur type. On peut maintenant définir les environnements (1.4) et la fonction de localisation (1.5).

$$\rho \in Env = Id \to \mathcal{NCP} \tag{1.4}$$

$$\rho(id) = \langle id, (), typeof(id) \rangle \tag{1.5}$$

1.2.2 Définition de la sémantique concrète du langage $\mathcal{L}_0$

Le domaine concret $\mathcal{D}$ de représentation de notre sémantique sera : un domaine numérique classique $\mathbb{R}$, pour les variables réelles et les constantes, auquel est ajoutée la représentation par chemin d'accès non constant $\mathcal{NCP}$, pour les pointeurs.

L'environnement d'évaluation des localisations est défini : $\sigma \in Store = \mathcal{NCP} \to \mathcal{D}$.

Ainsi, la sémantique concrète pour les expressions *Fig.* 1.2 est définie avec :

$$\mathbb{E}_{np} \in \langle expression \rangle \to Env \times Store \to \mathcal{D} \times \Omega$$

$$\mathbb{E}_{p} \in \langle expression \rangle \to Env \times Store \to \mathcal{NCP} \times \Omega$$

où $\mathbb{E}_{np}[\![expr]\!]\langle \rho, \sigma \rangle$ et $\mathbb{E}_{p}[\![expr]\!]\langle \rho, \sigma \rangle$ représentent l'évaluation d'une *expression expr*, avec la configuration $\langle \rho, \sigma \rangle$, dans respectivement le domaine numérique $\mathcal{D}$ ou le domaine des chemins d'accès non constant $\mathcal{NCP}$, accompagné ou non d'une erreur.

De même, la sémantique concrète pour les instructions *Fig.* 1.3 est définie avec :

$$\mathbb{S} \in \langle statement \rangle \to Env \times Store \times \Omega \to Env \times Store \times \Omega$$

où $\mathbb{S}[\![stat]\!]\langle \rho, \sigma, \mathcal{O} \rangle$ représente l'exécution d'un *statement stat* dans la configuration $\langle \rho, \sigma, \mathcal{O} \rangle$ et renvoie une nouvelle configuration.

De plus, du fait qu'on est typé, un abus de la notation let est fait lors de la récupération du type d'un chemin non constant et par la suite celui d'un chemin constant.

Seule la sémantique relative aux pointeurs est présentée. Celle concernant le domaine numérique est supposée connue, par exemple celle décrite dans la thèse de Creusillet [Cre96, Annexes C et D].

1.3 La sémantique abstraite du langage $\mathcal{L}_0$

1.3.1 L'ensemble des chemins d'accès constant $\mathcal{CP}^\#$ du langage $\mathcal{L}_0$

Le chemin d'accès non constant $\mathcal{NCP}$ sera abstrait par un ensemble de chemins d'accès constant $\mathcal{CP}^\#$. Un chemin d'accès constant permet d'accéder à une cellule mémoire ou à un ensemble de cellules mémoires. Comme $\mathcal{NCP}$, il sera composé d'un triplet et seul son deuxième élément change.

$$cp \in LOC^\# = \mathcal{CP}^\# = (Name \times V_{ref} \times Type) + \{\omega\} \tag{1.16}$$

avec

$$V_{ref} = \{\mathbb{N} \cup \{*\}\}^*$$

où $*$ représente tous les indices de la référence considérée.

Contrairement à $\mathcal{NCP}$, lorsqu'un chemin d'accès constant accède à une cellule mémoire précise, ce chemin est unique. Par exemple, le code 2, ci-dessous, donne les chemins non constant $\langle p, (0), int \rangle$ et $\langle t, (0), int \rangle$, et ils accèdent à la même cellule mémoire. Alors que dans l'ensemble $\mathcal{CP}^\#$, seul le chemin constant $\langle t, (0), int \rangle$ est présent, mais accompagné d'un graphe *points-to* présenté dans la section suivante.

```c
int t[10], *p;
p=&t[0];
```

CODE 2 – Exemple de différence entre $\mathcal{NCP}$ et $\mathcal{CP}^\#$

$id \in \langle id \rangle\,,\ V, V_1, V_2 \in \langle variable \rangle\,,\ e \in \langle expression \rangle\,,\ i \in \mathbb{N}$

$$
\begin{aligned}
\mathbb{E}_p[\![id]\!]\langle\rho,\sigma\rangle = {} & \langle \rho(id), \varnothing \rangle & (1.6)\\
\mathbb{E}_p[\![V.id]\!]\langle\rho,\sigma\rangle = {} & \text{if } typeof(V) = \text{struct}\{t_1, \dots, t_{i-1}, id : t, t_{i+1}, \dots, t_n\} & (1.7)\\
& \quad \text{let } \langle\langle var, seq, typeof(V)\rangle, \mathcal{O}\rangle = \mathbb{E}_p[\![V]\!]\langle\rho,\sigma\rangle \text{ in}\\
& \qquad \langle\langle var, seq.i, t\rangle, \mathcal{O}\rangle\\
& \text{else } \langle \emptyset, \omega \rangle\\
\mathbb{E}_p[\![V[i]]\!]\langle\rho,\sigma\rangle = {} & \text{if } typeof(V) = t[n] \text{ with } 0 \le i < n & (1.8)\\
& \quad \text{let } \langle\langle var, seq, t[n]\rangle, \mathcal{O}\rangle = \mathbb{E}_p[\![V]\!]\langle\rho,\sigma\rangle \text{ in } \langle\langle var, seq.i, t\rangle, \mathcal{O}\rangle\\
& \text{elseif } typeof(V) = \text{pointer}(t)\\
& \quad \mathbb{E}_p[\![V +_{ptr} i]\!]\langle\rho,\sigma\rangle\\
& \text{else } \langle \emptyset, \omega \rangle\\
\mathbb{E}_p[\![V[id]]\!]\langle\rho,\sigma\rangle = {} & \text{if } typeof(id) \in \langle int\text{-}type \rangle & (1.9)\\
& \quad \text{let } \langle j, \mathcal{O}\rangle = \mathbb{E}_{np}[\![id]\!]\langle\rho,\sigma\rangle \text{ in}\\
& \quad \text{let } \langle cp, \mathcal{O}'\rangle = \mathbb{E}_p[\![V[j]]\!]\langle\rho,\sigma\rangle \text{ in } \langle cp, \mathcal{O} \cup \mathcal{O}'\rangle\\
& \text{else } \langle \emptyset, \omega \rangle\\
\mathbb{E}_p[\![*V]\!]\langle\rho,\sigma\rangle = {} & \text{if } typeof(V) = \text{pointer}(t) \wedge V \in dom(\rho) & (1.10)\\
& \quad \text{let } \langle\langle var, seq, \text{pointer}(t)\rangle, \mathcal{O}\rangle = \mathbb{E}_p[\![V]\!]\langle\rho,\sigma\rangle \text{ in } \langle\langle var, seq.0, t\rangle, \mathcal{O}\rangle\\
& \text{elseif } typeof(V) = \text{pointer}(t)\\
& \qquad \langle ncp_{undefined}, \varnothing \rangle\\
& \text{else } \langle \emptyset, \omega \rangle\\
\mathbb{E}_p[\![\&V]\!]\langle\rho,\sigma\rangle = {} & \text{let } \langle\langle var, seq, t\rangle, \mathcal{O}\rangle = \mathbb{E}_p[\![V]\!]\langle\rho,\sigma\rangle \text{ in} & (1.11)\\
& \quad \langle\langle var.\text{“}\#address\text{”}, (), \text{pointer}(t)\rangle, \mathcal{O}\rangle \qquad \text{// création d'une cellule cf 1.3.3}\\
\mathbb{E}_p[\![V +_{ptr} e]\!]\langle\rho,\sigma\rangle = {} & \text{if } typeof(V) = \text{pointer}(t) \wedge typeof(e) \in \langle int\text{-}type \rangle & (1.12)\\
& \quad \text{let } \langle j, \mathcal{O}\rangle = \mathbb{E}_{np}[\![e]\!]\langle\rho,\sigma\rangle \text{ in}\\
& \qquad \text{if } \mathbb{E}_p[\![*V]\!]\langle\rho,\sigma\rangle = \langle ncp_{undefined}, \mathcal{O}'\rangle \wedge j = 0\\
& \qquad\quad \langle ncp_{undefined}, \mathcal{O} \cup \mathcal{O}'\rangle\\
& \qquad \text{elseif } \mathbb{E}_p[\![*V]\!]\langle\rho,\sigma\rangle = \langle\langle var, (i_1, \dots, i_{n-1}, i_n), t\rangle, \mathcal{O}'\rangle\\
& \qquad\quad \langle\langle var, (i_1, \dots, i_{n-1}, i_n + j), t\rangle, \mathcal{O} \cup \mathcal{O}'\rangle\\
& \qquad \text{else } \langle \emptyset, \omega \cup \mathcal{O}\rangle\\
& \text{else } \langle \emptyset, \omega \rangle\\
\mathbb{E}_{np}[\![V_1 -_{ptr} V_2]\!]\langle\rho,\sigma\rangle = {} & \text{let } \langle\langle var_1, seq_1.i, \text{pointer}(t_1)\rangle, \mathcal{O}\rangle = \mathbb{E}_p[\![V_1]\!]\langle\rho,\sigma\rangle \text{ in} & (1.13)\\
& \text{let } \langle\langle var_2, seq_2.j, \text{pointer}(t_2)\rangle, \mathcal{O}'\rangle = \mathbb{E}_p[\![V_2]\!]\langle\rho,\sigma\rangle \text{ in}\\
& \quad \text{if } var_1 = var_2 \wedge seq_1 = seq_2 \wedge t_1 = t_2\\
& \qquad \langle i - j, \mathcal{O} \cup \mathcal{O}'\rangle\\
& \quad \text{else } \langle \emptyset, \omega \rangle
\end{aligned}
$$

Figure 1.2 – Sémantique concrète des expressions du langage $\mathcal{L}_0$

$s_1, s_2 \in \langle statement \rangle\,,\ lhs \in \langle lhs \rangle\,,\ e \in \langle expression \rangle$

$$
\begin{aligned}
\mathbb{S}[\![s_1; s_2]\!]\langle\rho,\sigma,\mathcal{O}\rangle = {} & \mathbb{S}[\![s_2]\!]\langle \mathbb{S}[\![s_1]\!]\langle\rho,\sigma,\mathcal{O}\rangle\rangle & (1.14)\\
\mathbb{S}[\![lhs \leftarrow e]\!]\langle\rho,\sigma,\mathcal{O}\rangle = {} & \text{let } \langle ncp_{lhs}, \mathcal{O}_{lhs}\rangle = \mathbb{E}_p[\![lhs]\!]\langle\rho,\sigma\rangle \text{ in} & (1.15)\\
& \text{let } \langle v, \mathcal{O}_e\rangle = \mathbb{E}_{np}[\![e]\!]\langle\rho,\sigma\rangle \text{ in}\\
& \quad \langle\langle \rho, \sigma[ncp_{lhs} \mapsto v]\rangle, \mathcal{O} \cup \mathcal{O}_{lhs} \cup \mathcal{O}_e\rangle
\end{aligned}
$$

Figure 1.3 – Sémantique concrète des instructions du langage $\mathcal{L}_0$

On définit également *undefined* (1.17) et `NULL` (1.18) :

$$cp_{undefined} = ncp_{undefined} = \langle undefined, (), overloaded\rangle \tag{1.17}$$
$$cp_{\texttt{NULL}} = ncp_{\texttt{NULL}} = \langle \texttt{NULL}, (), overloaded\rangle \tag{1.18}$$

1.3.2 L'ensemble des graphes *points-to* $\mathcal{PT}^\#$

L'ensemble des graphes *points-to* (1.19) est défini par un ensemble de couples (origine, destination) qui signifie qu'une origine pointe vers une destination. L'origine doit être de type pointeur. L'origine et la destination sont représentées par les chemins d'accès constant $\mathcal{CP}^\#$ présentés précédemment en 1.3.1.

$$P \in \mathcal{PT}^\# = \mathcal{P}(\mathcal{CP}^\# \times \mathcal{CP}^\#) \cup \{\omega\} \tag{1.19}$$

Étant donné que les pointeurs sont typés, on doit également vérifier que l'arc (origine, destination) est bien typé, c'est-à-dire que le type pointé de l'origine est équivalent au type de destination. Cette relation d'équivalence de types est présentée dans l'équation (A.6) p. 25.

Chaque déclaration de variable de type pointeur engendre automatiquement la création d'un arc *points-to* de cette variable vers le chemin d'accès constant $cp_{undefined}$.

L'exemple du code 2 donne l'ensemble $\mathcal{CP}^\#$ $\{\langle t, (0), int\rangle, \langle p, (), \text{pointer}(int)\rangle, \ \dots\}$ et le graphe $\mathcal{PT}^\#$ $\{(\langle p, (), \text{pointer}(int)\rangle, \langle t, (0), int\rangle)\}$

1.3.3 Le contexte formel et le référencement (&)

Comme on a pu le voir dans (1.11), la localisation d'une variable nécessite parfois la création d'un nouvel identifiant abstrait. De même, les paramètres formels d'une fonction peuvent nécessiter la création d'arcs *points-to* formant notre contexte formel. On définit ainsi $Id^\#$ qui représente nos identifiants concrets auxquels sont rajoutés des identifiants abstraits.

Ainsi, on définit la fonction *newstub* (1.20) qui prendra en paramètre les éléments d'un chemin d'accès constant, c'est-à-dire un nom, une liste de références et un type, et qui renverra un chemin d'accès constant dans $\mathcal{CP}^\#$.

$$newstub : \left\{ \begin{array}{lcl} Id^\# \times \{\mathbb{N} \cup \{*\}\}^* \times \langle type\rangle & \rightarrow & \mathcal{CP}^\# \\ name, seq, type & \mapsto & \langle name, seq, type\rangle \end{array} \right. \tag{1.20}$$

Notre sémantique permet de générer le contexte formel à la demande pour éviter les explosions mémoire lors de notre analyse.

1.3.4 Création et destruction d'arc *points-to*

Une fonction permettant d'avoir l'ensemble des arcs *points-to* créés à partir de deux ensembles de cellules $\mathcal{CP}^\#$, représentant les sources pour le premier ensemble et les destinations pour le second, est nécessaire (1.21) [6]. De même, la définition d'une fonction permettant d'avoir l'ensemble des arcs à supprimer est faite (1.22) [6]. Ces deux fonctions correspondent aux ensembles Gen et Kill d'une analyse de flot de données comme celle du « Dragon Book » [ALSU06].

6. Ces opérateurs Kill et Gen effectuent des mises à jour fortes sur nos graphes $\mathcal{PT}^\#$. On les affinera par la suite avec la définition du treillis $\mathcal{CP}^\#$.

$$
\begin{aligned}
Gen : \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{P}(\mathcal{CP}^\#) &\rightarrow \mathcal{PT}^\# \\
Gen(Source, Destination) &= \text{if } Source = \emptyset \vee Destination = \emptyset \\
&\quad\ \text{then } \emptyset \\
&\quad\ \text{else } \{(s,d) | s \in Source \wedge d \in Destination\}
\end{aligned}
\tag{1.21}
$$

$$
\begin{aligned}
Kill : \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{PT}^\# &\rightarrow \mathcal{PT}^\# \\
Kill(Source, P) &= \text{if } Source = \emptyset \\
&\quad\ \text{then } \emptyset \\
&\quad\ \text{else } \{(s,d) | s \in Source \wedge (s,d) \in P\}
\end{aligned}
\tag{1.22}
$$

1.3.5 Définition de la sémantique abstraite du langage $\mathcal{L}_0$

On peut maintenant construire notre sémantique abstraite.

Les environnements de localisation abstraite seront : $Env^\# = \mathcal{P}(Id \rightarrow \mathcal{CP}^\#)$. Les environnements d'évaluation abstraite évaluent un couple $\mathcal{CP}^\# \times \mathcal{PT}^\#$ dans le domaine numérique abstrait $\mathcal{D}^\#$: $Store^\# = \mathcal{P}(\mathcal{CP}^\# \times \mathcal{PT}^\# \rightarrow \mathcal{D}^\#)$.

On suppose que l'on dispose d'un domaine numérique abstrait $\mathcal{D}^\#$ connu. Par exemple, une abstraction par intervalles comme celle introduite par Cousot et Cousot [CC76], ou une abstraction polyédrique [CH78]. On ne redétaille donc pas la sémantique dans ce domaine.

Ainsi, la sémantique abstraite pour les expressions *Fig.* 1.4 est définie avec :

$$\mathbb{E}^\#_{np} \in \langle expression \rangle \rightarrow Env^\# \times Store^\# \times \mathcal{PT}^\# \rightarrow \mathcal{P}(\mathcal{D}^\#) \times \mathcal{PT}^\# \times \Omega$$

$$\mathbb{E}^\#_{p} \in \langle expression \rangle \rightarrow Env^\# \times Store^\# \times \mathcal{PT}^\# \rightarrow \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{PT}^\# \times \Omega$$

où $\mathbb{E}^\#_{np}[\![expr]\!]\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P \right\rangle$ et $\mathbb{E}^\#_{p}[\![expr]\!]\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P \right\rangle$ représentent l'évaluation d'une *expression expr*, avec la configuration abstraite $\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P \right\rangle$, dans respectivement le domaine numérique $\mathcal{D}^\#$ ou le domaine des chemins d'accès non constant $\mathcal{CP}^\#$, accompagné d'un nouveau graphe $\mathcal{PT}^\#$ et de la présence ou de l'absence d'erreurs.

De même, la sémantique abstraite pour les instructions *Fig.* 1.5 est définie avec :

$$\mathbb{S}^\# \in \langle statement \rangle \rightarrow Env^\# \times Store^\# \times \mathcal{PT}^\# \times \Omega \rightarrow Env^\# \times Store^\# \times \mathcal{PT}^\# \times \Omega$$

où $\mathbb{S}^\#[\![stat]\!]\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O} \right\rangle$ représente l'exécution d'un *statement stat* dans la configuration abstraite $\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O} \right\rangle$ et renvoie une nouvelle configuration abstraite.

Pour la lecture de la sémantique, on définit :

$$\text{foreach } Prop \text{ do } Inst \equiv \bigcup_{Prop}^\# Inst \quad \text{où } Prop \text{ peut être implicitement appliqué sur un élément valide.}$$

$$\langle cp_1, P_1, \mathcal{O}_1 \rangle \cup^\# \langle cp_2, P_2, \mathcal{O}_2 \rangle \equiv \langle cp_1 \cup^{\#7} cp_2, P_1 \cup^\# P_2, \mathcal{O}_1 \cup^\# \mathcal{O}_2 \rangle$$

1.4 Exemple

L'exemple du code 3 permet d'initialiser les champs pointeurs d'une structure. La structure est composée de pointeurs de pointeurs et le travail est fait sur un pointeur vers cette structure. Dans un premier temps, (S2) initialisera un pointeur avant d'initialiser la structure (S3-S5).

Par abus de notation et pour plus de clarté, on notera $\langle type \rangle *$pour représenter pointer($\langle type \rangle$).

```
struct bar {int **pp1; int **pp2;};

void foo(struct bar *ps) {
  int i = 1, j = 2, *p;       //S1
```

7. $\cup^\#$ correspond à la borne supérieur pour les $\mathcal{CP}^\#$ *Sec.* 2.3

$id \in \langle id \rangle\,,\ V, V_1, V_2 \in \langle variable \rangle\,,\ e \in \langle expression \rangle\,,\ i \in \mathbb{N}$
$\mathcal{X}^\# = \left\langle \mathcal{R}^\#, \mathcal{S}^\#, P \right\rangle \in Env^\# \times Store^\# \times \mathcal{PT}^\#$

$$\begin{aligned}
\mathbb{E}_p^\#[\![id]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{foreach } \rho \in \mathcal{R}^\# \text{ do} && (1.23)\\
& \quad \text{if } id \in \langle Id_{\texttt{FORMAL}} \rangle \\
& \quad\quad \text{let } source = \langle var, seq, typeof(id) \rangle = \rho(id) \text{ in} \\
& \quad\quad\quad \text{if } \exists (source, dest) \in P \text{ then } \langle source, P, \varnothing \rangle \\
& \quad\quad\quad \text{else let } cp = newstub(var, seq.0, t) \text{ in } \langle source, P \cup \{(source, cp)\}, \varnothing \rangle \\
& \quad \text{else } \langle \rho(id), P, \varnothing \rangle \\
\mathbb{E}_p^\#[\![V.id]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{if } typeof(V) = \text{struct}\{\dots, t_{i-1}, id : t, t_{i+1}, \dots\} && (1.24)\\
& \quad \text{foreach } \left\langle \langle var, seq, typeof(V) \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do } \left\langle \langle var, seq.i, t \rangle, P', \mathcal{O} \right\rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \rangle \\
\mathbb{E}_p^\#[\![V[i]]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{if } typeof(V) \in t[n] \text{ with } 0 \le i < n && (1.25)\\
& \quad \text{foreach } \left\langle \langle var, seq, t[n] \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do } \left\langle \langle var, seq.i, t \rangle, P', \mathcal{O} \right\rangle \\
& \text{elseif } typeof(V) \in \text{pointer}(t) \text{ then } \mathbb{E}_p^\#[\![V +_{ptr} i]\!]\left\langle \mathcal{X}^\# \right\rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \rangle \\
\mathbb{E}_p^\#[\![V[id]]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{if } typeof(id) \in \langle int\text{-}type \rangle \wedge && (1.26)\\
& \quad (typeof(V) = t[n] \vee typeof(V) = \text{pointer}(t)) \\
& \quad \text{foreach } \left\langle \langle var, seq, typeof(V) \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do } \left\langle \langle var, seq.*, t \rangle, P', \mathcal{O} \right\rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \rangle \\
\mathbb{E}_p^\#[\![*V]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{if } typeof(V) \in \text{pointer}(t) \wedge V \in dom(\mathcal{R}^\#) && (1.27)\\
& \quad \text{foreach } \left\langle \langle var, seq, \text{pointer}(t) \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do} \\
& \quad \text{let } source = \langle var, seq, \text{pointer}(t) \rangle \text{ in} \\
& \quad\quad \text{if } \exists (source, dest) \in P' \text{ then } \left\langle dest, P', \mathcal{O} \right\rangle \\
& \quad\quad \text{else let } cp = newstub(var, seq.0, t) \text{ in } \left\langle cp, P' \cup \{(source, cp)\}, \mathcal{O} \right\rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \rangle \\
\mathbb{E}_p^\#[\![\&V]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{foreach } \left\langle \langle var, seq, t \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do} && (1.28)\\
& \text{let } dest = \langle var, seq, t \rangle \text{ in} \\
& \quad \text{if } \exists (s, dest) \in P' | s = \langle var.\text{“}\#address\text{”}, (), \text{pointer}(t) \rangle \text{ then } \left\langle s, P', \mathcal{O} \right\rangle \\
& \quad \text{else let } cp = newstub(var.\text{“}\#address\text{”}, (), \text{pointer}(t)) \text{ in } \left\langle cp, P' \cup \{(cp, dest)\}, \mathcal{O} \right\rangle \\
\mathbb{E}_p^\#[\![V +_{ptr} e]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{if } typeof(V) = \text{pointer}(t) \wedge typeof(e) \in \langle int\text{-}type \rangle && (1.29)\\
& \quad \text{foreach } \left\langle \langle var, seq.ref, t \rangle, P', \mathcal{O} \right\rangle \in \mathbb{E}_p^\#[\![*V]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do} \\
& \quad \text{if } \mathbb{E}_{np}^\#[\![e]\!]\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P' \right\rangle = \left\langle \{j\}, P'', \mathcal{O}' \right\rangle \text{ then } \left\langle \langle var, seq.(ref + j), t \rangle, P'', \mathcal{O} \cup \mathcal{O}' \right\rangle \\
& \quad \text{else } \left\langle \langle var, seq.*, t \rangle, P', \mathcal{O} \right\rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \rangle \\
\mathbb{E}_{np}^\#[\![V_1 -_{ptr} V_2]\!]\left\langle \mathcal{X}^\# \right\rangle = \ & \text{foreach } \langle \langle var_1, seq_1.i, \text{ptr}(t_1) \rangle, P_1, \mathcal{O}_1 \rangle \in \mathbb{E}_p^\#[\![V_1]\!]\left\langle \mathcal{X}^\# \right\rangle \text{ do} && (1.30)\\
& \text{foreach } \langle \langle var_2, seq_2.j, \text{ptr}(t_2) \rangle, P_2, \mathcal{O}_2 \rangle \in \mathbb{E}_p^\#[\![V_2]\!]\left\langle \mathcal{R}^\#, \mathcal{S}^\#, P_1 \right\rangle \text{ do} \\
& \text{if } var_1 = var_2 \wedge seq_1 = seq_2 \wedge t_1 = t_2 \text{ then } \langle i - j, P_2, \mathcal{O}_1 \cup \mathcal{O}_2 \rangle \\
& \text{else } \langle \{\}, \emptyset, \omega \cup \mathcal{O}_1 \cup \mathcal{O}_2 \rangle
\end{aligned}$$

FIGURE 1.4 – Sémantique abstraite des expressions du langage $\mathcal{L}_0$

$s_1, s_2 \in \langle statement \rangle$, $lhs \in \langle lhs \rangle$, $e \in \langle expression \rangle$

$$\mathbb{S}^\#[\![s_1; s_2]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O} \rangle = \mathbb{S}^\#[\![s_2]\!]\langle \mathbb{S}^\#[\![s_1]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O} \rangle \rangle \tag{1.31}$$

$$\begin{aligned}
\mathbb{S}^\#[\![lhs \leftarrow e]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O} \rangle = \; & \text{foreach } \langle cp_{lhs}, P_1, \mathcal{O}_1 \rangle \in \mathbb{E}_p^\#[\![lhs]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P \rangle \text{ do} \\
& \text{foreach } \langle cp_e, P_2, \mathcal{O}_2 \rangle \in \mathbb{E}_p^\#[\![e]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P_1 \rangle \text{ do} \\
& \text{let } \mathcal{O}_{out} = \mathcal{O} \cup \mathcal{O}_1 \cup \mathcal{O}_2 \text{ in} \\
& \text{if } typeof(lhs) = typeof(e) \\
& \quad \text{foreach } (\rho, \sigma) \in (\mathcal{R}^\#, \mathcal{S}^\#) \text{ do} \\
& \quad \text{if } typeof(lhs) = \text{pointer}(type) \\
& \quad\quad \text{let } CP_{gen} = \{cp_{gen} | (cp_e, cp_{gen}) \in P_2\} \text{ in} \\
& \quad\quad \text{if } cp_{undefined} \notin CP_{gen} \wedge cp_{lhs} \neq cp_{undefined} \wedge cp_{lhs} \neq cp_{\texttt{NULL}} \\
& \quad\quad\quad \text{let } Kill = Kill(\{cp_{lhs}\}, P) \text{ in} \\
& \quad\quad\quad \text{let } Gen = Gen(\{cp_{lhs}\}, CP_{gen}) \text{ in} \\
& \quad\quad\quad\quad \langle \rho, \sigma[(cp_{lhs}, _) \mapsto \sigma(cp_e, _)], P_2 \setminus^\# Kill \cup^\# Gen, \mathcal{O}_{out} \rangle \\
& \quad\quad \text{else } \langle \emptyset, \emptyset, \emptyset, \omega \cup \mathcal{O}_{out} \rangle \\
& \quad \text{else } \langle \rho, \sigma[(cp_{lhs}, _) \mapsto \sigma(cp_e, _)], P_2, \mathcal{O}_{out} \rangle \\
& \text{else } \langle \emptyset, \emptyset, \emptyset, \omega \cup \mathcal{O}_{out} \rangle
\end{aligned} \tag{1.32}$$

FIGURE 1.5 – Sémantique abstraite des instructions du langage $\mathcal{L}_0$

```
  p = &j;                   //S2
  *((*ps).pp1) = &i;        //S3
  *((*ps).pp2) = p;         //S4
  (*ps).pp1 = (*ps).pp2;    //S5
}
```

CODE 3 – Exemple pour la sémantique du langage $\mathcal{L}_0$

À la fin de notre analyse, nous obtenons les propriétés suivantes :

$$P = \{ (cp_{ps}, cp_{ps[0]}),\ (cp_{ps[0].pp1}, cp_{ps[0].pp2[0]}),\ (cp_{ps[0].pp2}, cp_{ps[0].pp2[0]}),\ (cp_{\&i}, cp_i),\ (cp_{ps[0].pp1[0]}, cp_i),\ (cp_{ps[0].pp2[0]}, cp_j),\ (cp_{\&j}, cp_j),\ (cp_p, cp_j)\}$$

$$\begin{aligned}
\sigma(cp_i, P) &= 1 \\
\sigma(cp_j, P) &= 2 \\
\sigma(cp_{ps[0].pp1[0]}, P) &= \sigma(cp_{\&i}, P) \\
\sigma(cp_{ps[0].pp2[0]}, P) &= \sigma(cp_p, P) = \sigma(cp_{\&j}, P) \\
\sigma(cp_{ps[0].pp1}, P) &= \sigma(cp_{ps[0].pp2}, P)
\end{aligned}$$

Le détail de l'analyse est présenté en annexe B.

2 Le langage $\mathcal{L}_1$: les conditions

2.1 La syntaxe de $\mathcal{L}_1$

Le langage $\mathcal{L}_0$ nous permet d'avoir une syntaxe de base permettant d'exécuter des programmes purement séquentiels. Mais pour avoir un programme intéressant, il faut pouvoir exécuter des conditions et des boucles. Ainsi, on élargit le langage $\mathcal{L}_0$ au langage $\mathcal{L}_1$ en ajoutant la possibilité de faire des tests et des boucles. Seule la syntaxe pour les $\langle statement \rangle$ change, *Fig.* 2.1.

$$
\begin{array}{lcl}
\langle \mathit{cond} \rangle & ::= & \langle \mathit{expression} \rangle \ \langle \mathit{relational\text{-}op} \rangle \ \langle \mathit{expression} \rangle \\
\langle \mathit{statement} \rangle & ::= & \langle \mathit{lhs} \rangle \leftarrow \langle \mathit{expression} \rangle \\
 & | & \text{if } \langle \mathit{cond} \rangle \text{ then } \langle \mathit{statement} \rangle \text{ [else } \langle \mathit{statement} \rangle \text{]} \\
 & | & \text{while } \langle \mathit{cond} \rangle \text{ do } \langle \mathit{statement} \rangle \\
 & | & \langle \mathit{statement} \rangle ; \langle \mathit{statement} \rangle
\end{array}
$$

Figure 2.1: Syntaxe des statements de $\mathcal{L}_1$

2.2 L'approximation

L'introduction des conditions implique l'apparition d'approximations. En effet, avec les conditions, un pointeur **peut** pointer vers une valeur (`MAY`), un pointeur pointe vers une valeur (`EXACT`), ou un pointeur **doit** pointer vers une valeur (`MUST`). `MUST` et `MAY` correspondent respectivement à des sur- et sous-approximation. En pratique, seul `EXACT` et `MAY` seront utilisés car `MUST` est difficilement calculable.

Ainsi, nous enrichissons notre ensemble *points-to* $\mathcal{PT}^{\#}$ avec cette approximation :

$$\mathcal{PT}^{\#} = (\mathcal{P}(\mathcal{CP}^{\#} \times \mathcal{CP}^{\#}) \times \{\mathtt{MAY}, \mathtt{EXACT}\}) \cup \{\omega\} \tag{2.1}$$

Par souci de clarté, on décomposera parfois notre ensemble P de *points-to* en deux sous-ensembles $P_{\mathtt{MAY}}$ et $P_{\mathtt{EXACT}}$ représentant respectivement l'ensemble d'arc $\mathcal{CP}^{\#}$ d'approximation `MAY` et `EXACT`.

$$P = \langle P_{\mathtt{MAY}}, \mathtt{MAY} \rangle \cup \langle P_{\mathtt{EXACT}}, \mathtt{EXACT} \rangle \qquad \text{avec } \langle P_\alpha, \alpha \rangle \cup \omega = \omega \cup \langle P_\alpha, \alpha \rangle = \omega$$

L'union $\cup^{\#}$ entre deux ensemble $\mathcal{PT}^{\#}$ est également redéfinie (2.2).

$$
\begin{aligned}
P_1 \cup^{\#} P_2 = \ & \text{let } P_{\mathtt{EXACT}} = P_{1_{\mathtt{EXACT}}} \cap P_{2_{\mathtt{EXACT}}} \text{ in} \\
& \quad \text{let } P_{\mathtt{MAY}} = (P_{1_{\mathtt{MAY}}} \cup P_{2_{\mathtt{MAY}}} \cup P_{1_{\mathtt{EXACT}}} \cup P_{2_{\mathtt{EXACT}}}) \setminus P_{\mathtt{EXACT}} \text{ in} \\
& \quad \langle P_{\mathtt{EXACT}}, \mathtt{EXACT} \rangle \cup \langle P_{\mathtt{MAY}}, \mathtt{MAY} \rangle
\end{aligned} \tag{2.2}
$$

2.3 Les relations d'ordre pour $\mathcal{CP}^{\#}$

Nous devons également préciser quelles sont les relations d'ordre pour notre ensemble $\mathcal{CP}^{\#}$ introduit en 1.3.1 de manière à ce que cet ensemble soit bien un treillis.

Le treillis $\mathcal{CP}^{\#}$ se décrit à partir de ses trois composantes, *Fig.* 2.2. Celles-ci sont également des treillis dont la structure est décrite en annexe A. En plus des relations classiques $\cup^{\#}$, $\cap^{\#}$ et $\subseteq^{\#}$ [8], deux nouvelles relations sont définies : $\mathit{kill}_{\mathtt{MAY}}$ ou $\mathit{kill}_{\mathtt{M}}$, et $\mathit{kill}_{\mathtt{EXACT}}$ ou $\mathit{kill}_{\mathtt{E}}$. Elles permettent de savoir si un $\mathcal{CP}^{\#}$ peut « tuer » un autre $\mathcal{CP}^{\#}$.

$$
\begin{aligned}
cp_1 \cup^{\#} cp_2 &= \left\langle Name_1 \cup^{\#} Name_2, Ref_1 \cup^{\#} Ref_2, Type_1 \cup^{\#} Type_2 \right\rangle \\
cp_1 \cap^{\#} cp_2 &= \left\langle Name_1 \cap^{\#} Name_2, Ref_1 \cap^{\#} Ref_2, Type_1 \cap^{\#} Type_2 \right\rangle \\
cp_1 \subseteq^{\#} cp_2 &= Name_1 \subseteq^{\#} Name_2 \wedge Ref_1 \subseteq^{\#} Ref_2 \wedge Type_1 \subseteq^{\#} Type_2 \\
cp_1 \ \mathit{kill}_{\mathtt{M}} \ cp_2 &= Name_1 \ \mathit{kill}_{\mathtt{M}} \ Name_2 \wedge Ref_1 \ \mathit{kill}_{\mathtt{M}} \ Ref_2 \wedge Type_1 \ \mathit{kill}_{\mathtt{M}} \ Type_2 \\
cp_1 \ \mathit{kill}_{\mathtt{E}} \ cp_2 &= Name_1 \ \mathit{kill}_{\mathtt{E}} \ Name_2 \wedge Ref_1 \ \mathit{kill}_{\mathtt{E}} \ Ref_2 \wedge Type_1 \ \mathit{kill}_{\mathtt{E}} \ Type_2
\end{aligned}
$$

Figure 2.2 – Relations sur le treillis $\mathcal{CP}^{\#}$

Un prédicat d'atomicité est également défini (2.3) permettant de savoir si un $\mathcal{CP}^{\#}$ pointe vers une unique cellule ou un ensemble de cellules. On renvoie vrai dans le premier cas et faux dans le second.

$$
\mathit{atomic} : \left\{ \begin{array}{ccc} \mathcal{CP}^{\#} & \rightarrow & \mathit{bool} \\ \langle name, seq, type \rangle & \mapsto & * \notin seq \end{array} \right. \tag{2.3}
$$

8. $\cup^{\#}$, $\cap^{\#}$ et $\subseteq^{\#}$ correspondent à la borne supérieure, la borne inférieure et l'inclusion dans les treillis.

2.4 L'affectation

La mise en place de l'approximation sur les pointeurs nécessite une raffinement de notre affectation. Ainsi, une redéfinition des ensembles Kill et Gen doit être faite.

Calcul de l'ensemble *Kill* L'ensemble $Kill$ est décomposé en deux sous-ensembles $Kill_1$ et $Kill_2$. Le premier sous-ensemble $Kill_1$ (2.4) correspond aux arcs $\mathcal{PT}^\#$ qui seront supprimés définitivement. Le second $Kill_2$ (2.5) correspond aux arcs `EXACT` dont l'approximation doit passer à `MAY`.

$$\begin{array}{rcl} Kill_1 : \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{PT}^\# & \rightarrow & \mathcal{PT}^\# \\ Kill_1(Source, P) & = & \text{if } Source = \{s\} \wedge atomic(s) \\ & & \quad \text{then } \{(s,d,a) | (s,d,a) \in P\} \\ & & \quad \text{else } \{\} \end{array} \tag{2.4}$$

$$\begin{array}{rcl} Kill_2 : \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{PT}^\# & \rightarrow & \mathcal{PT}^\# \\ Kill_2(Source, P_{\mathtt{EXACT}}) & = & \{(cp_{die}, d) | \exists s \in Source, s\ kill_{\mathtt{M}}\ cp_{die}\} \setminus^\# Kill_1(Source, P_{\mathtt{EXACT}}) \end{array} \tag{2.5}$$

Calcul de l'ensemble *Gen* Comme l'ensemble $Kill$, on décompose l'ensemble Gen en deux sous-ensembles. Gen_1 (2.6) correspond aux arcs détruit par $Kill_2$ dont on doit faire passer l'approximation à `MAY`. Gen_2 (2.7) correspond aux arcs que l'on souhaite créer entre deux ensembles $\mathcal{CP}^\#$.

$$\begin{array}{rcl} Gen_1 : \mathcal{PT}^\# & \rightarrow & \mathcal{PT}^\# \\ Gen_1(P) & = & \{(s,d,\mathtt{MAY}) | (s,d,a) \in P\} \end{array} \tag{2.6}$$

$$\begin{array}{rl} Gen_2 : \mathcal{P}(\mathcal{CP}^\#) \times \mathcal{P}(\mathcal{CP}^\#) \rightarrow \mathcal{PT}^\# & \\ Gen_2(Source, Dest) = & \text{if } Source = \{s\} \wedge Dest = \{d\} \wedge atomic(s) \wedge atomic(d) \\ & \quad \{(s,d,\mathtt{EXACT})\} \\ & \text{else foreach } s \in Source, d \in Dest \text{ do} \\ & \quad \{(s,d,\mathtt{MAY})\} \end{array} \tag{2.7}$$

Sémantique abstraite de l'affectation La sémantique abstraite de l'affectation sera la même que celle vue en (1.32) à la différence des ensembles $Kill$ et Gen qui ont été redéfinis précédemment. On ne montrera que la partie qui diffère en (2.8).

$$\begin{array}{l} \mathbb{S}^\#[\![lhs \leftarrow e]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O}\rangle = [\ldots] \\ \qquad \text{if } cp_{undefined} \notin CP_{gen} \wedge cp_{lhs} \neq cp_{undefined} \wedge cp_{lhs} \neq cp_{\mathtt{NULL}} \\ \qquad\quad \text{let } K_1 = Kill_1(\{cp_{lhs}\}, P) \text{ in} \\ \qquad\quad \text{let } K_2 = Kill_2(\{cp_{lhs}\}, P_{\mathtt{EXACT}}) \text{ in} \\ \qquad\quad \text{let } G_1 = Gen_1(Kill_2) \text{ in} \\ \qquad\quad \text{let } G_2 = Gen_2(\{cp_{lhs}\}, CP_{gen}) \text{ in} \\ \qquad\qquad \langle \rho, \sigma[(cp_{lhs}, \_) \mapsto \sigma(cp_e, \_)], P_2 \setminus K_1 \setminus K_2 \cup G_1 \cup G_2, \mathcal{O}_{out}\rangle \\ \qquad [\ldots] \end{array} \tag{2.8}$$

FIGURE 2.3 – Sémantique abstraite de l'affectation

2.5 Le test

Relation entre $\mathcal{CP}^\#$ Pour simplifier notre sémantique pour le test, les relations entre $\mathcal{CP}^\#$ possible est définie, *Fig.* 2.4. Quelle que soit la comparaison entre références considérée, si l'une des références vaut $*$ alors la comparaison est vraie.

$$
\begin{aligned}
\langle v_1, s_1.r_1, t_1\rangle == \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 = v_2 \wedge s_1 = s_2 \wedge t_1 = t_2 \wedge r_1 = r_2\\
\langle v_1, s_1.r_1, t_1\rangle \neq \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 \neq v_2 \vee s_1 \neq s_2 \vee t_1 \neq t_2 \vee r_1 \neq r_2\\
\langle v_1, s_1.r_1, t_1\rangle < \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 = v_2 \wedge s_1 = s_2 \wedge t_1 = t_2 \wedge r_1 < r_2\\
\langle v_1, s_1.r_1, t_1\rangle > \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 = v_2 \wedge s_1 = s_2 \wedge t_1 = t_2 \wedge r_1 > r_2\\
\langle v_1, s_1.r_1, t_1\rangle \leq \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 = v_2 \wedge s_1 = s_2 \wedge t_1 = t_2 \wedge r_1 \leq r_2\\
\langle v_1, s_1.r_1, t_1\rangle \geq \langle v_2, s_2.r_2, t_2\rangle &\equiv v_1 = v_2 \wedge s_1 = s_2 \wedge t_1 = t_2 \wedge r_1 \geq r_2
\end{aligned}
$$

FIGURE 2.4 – Relation entre $\mathcal{CP}^{\#}$

Sémantique abstraite du test À partir de ces relations, la sémantique pour les conditions est donnée (2.9). On ne considère que le cas où les expressions testées sont des pointeurs, `NULL` ou *undefined*. Dans le cas contraire, une analyse sans pointeur peut être effectuée pour la condition. Enfin, la sémantique pour le test est donnée (2.10).

$e_1, e_2 \in \langle expression \rangle$, $s_1, s_2 \in \langle statement \rangle$
$c = e_1 \, \langle relational\text{-}op \rangle \, e_2$
$\mathcal{Y}^{\#} \in Env^{\#} \times Store^{\#} \times \mathcal{PT}^{\#} \times \Omega$

$$
\begin{aligned}
\mathbb{S}^{\#}[\![c]\!]\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P, \mathcal{O}\rangle = {} & \text{foreach } \langle cp_1, P_1, \mathcal{O}_1\rangle \in \mathbb{E}_p^{\#}[\![e_1]\!]\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P\rangle \text{ do}\\
& \text{foreach } \langle cp_2, P_2, \mathcal{O}_2\rangle \in \mathbb{E}_p^{\#}[\![e_2]\!]\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P_1\rangle \text{ do}\\
& \text{let } \mathcal{O}_{out} = \mathcal{O} \cup \mathcal{O}_1 \cup \mathcal{O}_2 \text{ in}\\
& \quad \text{if } cp_1 = cp_{undefined} \vee cp_2 = cp_{undefined} \text{ then } \langle \emptyset, \emptyset, \emptyset, \omega \cup \mathcal{O}_{out}\rangle\\
& \quad \text{elseif } cp_1 = cp_{\mathtt{NULL}} \wedge cp_2 = cp_{\mathtt{NULL}} \text{ then } \langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P_2, \mathcal{O}_{out}\rangle\\
& \quad \text{elseif } cp_1 = cp_{\mathtt{NULL}} \vee cp_2 = cp_{\mathtt{NULL}} \text{ then } \langle \emptyset, \emptyset, \emptyset, \mathcal{O}_{out}\rangle\\
& \quad \text{else}\\
& \qquad \text{foreach } cp_1' | ((cp_1, cp_1') \in P_2) \vee (\exists cp, cp_1 kill\ cp \wedge (cp, cp_1') \in P_2) \text{ do}\\
& \qquad \text{foreach } cp_2' | ((cp_2, cp_2') \in P_2) \vee (\exists cp, cp_2 kill\ cp \wedge (cp, cp_2') \in P_2) \text{ do}\\
& \qquad \text{if } cp_1' \, \langle relational\text{-}op \rangle \, cp_2' \text{ then } \langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P_2, \mathcal{O}_{out}\rangle\\
& \qquad \text{else } \langle \emptyset, \emptyset, \emptyset, \mathcal{O}_{out}\rangle
\end{aligned}
\tag{2.9}
$$

$$
\mathbb{S}^{\#}[\![\text{if } c \text{ then } s_1 \text{ else } s_2]\!]\langle \mathcal{Y}^{\#}\rangle = \mathbb{S}^{\#}[\![s_1]\!]\langle \mathbb{S}^{\#}[\![c]\!]\langle \mathcal{Y}^{\#}\rangle\rangle \cup^{\#} \mathbb{S}^{\#}[\![s_2]\!]\langle \mathbb{S}^{\#}[\![!c]\!]\langle \mathcal{Y}^{\#}\rangle\rangle \tag{2.10}
$$

FIGURE 2.5 – Sémantique abstraite de la condition et du test

2.6 La boucle

Comme dans les parties précédentes, on ne détaillera pas la sémantique dans le domaine numérique. Par contre, dans cette sémantique, on n'utilise pas d'opération d'élargissement ∇ comme c'est traditionnellement fait, mais on réalise une sur-approximation de la fermeture transitive de la fonction de transition [ACI10].

Calcul du plus petit point fixe Le calcul du plus petit point fixe se fait de façon brute. Ainsi, on considère :

$$
\begin{aligned}
\mathcal{W} &= \text{while } c \text{ do } s;\\
&= \text{if } c \text{ then } (s;\ \text{while } c \text{ do } s;)\\
&= \text{if } c \text{ then } (s;\ \mathcal{W})
\end{aligned}
\tag{2.11}
$$

On itère notre relation (2.11) jusqu'à ce qu'elle soit stable, c'est-à-dire atteigne un point fixe qui sera le plus petit point fixe, notée *lfp*.
Cette méthode d'obtention du point fixe peut s'apparenter à un élargissement retardé [Min13] : le calcul de l'ensemble $\mathcal{PT}^{\#}$ se ferait par l'opérateur d'union abstrait $\cup^{\#}$ appliqué un nombre fini de fois.

Finitude du calcul du point fixe Pour assurer la finitude du calcul du point fixe, on définit la longueur maximale d'un chemin dans le graphe $\mathcal{PT}^{\#}$. Si cette longueur maximale est atteinte alors on remplace la destination du dernier arc par le dernier élément du chemin de même type, pour un chemin provenant du contexte formel, par $*$`HEAP`$*$, pour un élément du tas qui est introduit dans la partie suivante, *Sec.* 3, ou par $*anywhere*$ dans les autres cas. Ceci aura pour effet de créer un cycle à la fin du graphe et ainsi de limiter le nombre de nœuds du graphe $\mathcal{PT}^{\#}$. De plus, le nombre d'arcs dans $\mathcal{PT}^{\#}$ est également borné car ce n'est pas un multigraphe. Ainsi, on est assuré que notre graphe $\mathcal{PT}^{\#}$ est borné et fini.

De plus, pour accélérer le calcul de ce point fixe, on définit un nombre maximal d'arcs sortants [9] lors du calcul du point fixe pour une boucle sur le graphe $\mathcal{PT}^{\#}$. Lorsque ce nombre maximal est atteint, on remplace la destination du dernier arc formé par la borne supérieure de tous les $\mathcal{CP}^{\#}$ de destination.

Sémantique abstraite de la boucle La sémantique abstraite pour une boucle correspond à l'obtention du plus petit point fixe (2.14). (2.12) et (2.13) décrivent la démarche suivie pour trouver ce plus petit point fixe.

$e_1, e_2 \in \langle expression \rangle\,,\ s \in \langle statement \rangle$
$c = e_1\ \langle relational\text{-}op \rangle\ e_2$
$\mathcal{Y}^{\#} = \left\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P, \mathcal{O} \right\rangle \in Env^{\#} \times Store^{\#} \times \mathcal{PT}^{\#} \times \Omega$

$$\mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!]\langle\!\langle \mathcal{Y}^{\#} \rangle\!\rangle = \mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!]\langle\!\langle \mathbb{S}^{\#}[\![s]\!]\langle\!\langle \mathbb{S}^{\#}[\![c]\!]\langle\!\langle \mathcal{Y}^{\#} \rangle\!\rangle\rangle\!\rangle\rangle\!\rangle \cup^{\#} \mathbb{S}^{\#}[\![!c]\!]\langle\!\langle \mathcal{Y}^{\#} \rangle\!\rangle \tag{2.12}$$

$$\begin{aligned}\mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!] &= \lambda\kappa^{\#}.\mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!]\langle\!\langle \mathbb{S}^{\#}[\![s]\!]\langle\!\langle \mathbb{S}^{\#}[\![c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle\rangle\!\rangle\rangle\!\rangle \cup^{\#} \mathbb{S}^{\#}[\![!c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle \\ &= \left(\lambda f.\lambda\kappa^{\#}.f(\mathbb{S}^{\#}[\![s]\!]\langle\!\langle \mathbb{S}^{\#}[\![c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle\rangle\!\rangle) \cup^{\#} \mathbb{S}^{\#}[\![!c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle\right)\left(\mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!]\right)\end{aligned} \tag{2.13}$$

$$\mathbb{S}^{\#}[\![\text{while } c \text{ do } s]\!]\langle\!\langle \mathcal{Y}^{\#} \rangle\!\rangle = \ lfp\left(\lambda f.\lambda\kappa^{\#}.f(\mathbb{S}^{\#}[\![s]\!]\langle\!\langle \mathbb{S}^{\#}[\![c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle\rangle\!\rangle) \cup^{\#} \mathbb{S}^{\#}[\![!c]\!]\langle\!\langle \kappa^{\#} \rangle\!\rangle\right)\langle\!\langle \mathcal{Y}^{\#} \rangle\!\rangle \tag{2.14}$$

FIGURE 2.6 – Sémantique abstraite de la boucle

3 Le langage $\mathcal{L}_2$: l'allocation dynamique

3.1 La syntaxe de $\mathcal{L}_2$

Avec $\mathcal{L}_1$, on dispose du nécessaire pour pouvoir analyser plusieurs programmes simples. Mais le langage `C` offre la possibilité de faire de l'allocation mémoire dynamique. Cette dernière est l'une des motivations importantes pour une analyse de pointeurs. Ainsi, nous enrichissons notre langage $\mathcal{L}_1$ pour obtenir $\mathcal{L}_2$ qui nous permet de gérer l'allocation dynamique à l'aide des routines malloc et free. Notre nouvelle syntaxe est présentée ci-dessous, *Fig.* 3.1.

$$\begin{array}{lcll}\langle statement \rangle & ::= & \langle lhs \rangle \leftarrow \langle expression \rangle & \\ & | & \text{if } \langle cond \rangle \text{ then } \langle statement \rangle\ [\text{else } \langle statement \rangle] & \\ & | & \text{while } \langle cond \rangle \text{ do } \langle statement \rangle & \\ & | & \langle statement \rangle;\ \langle statement \rangle & \\ & | & \text{malloc}(\langle lhs \rangle,\ n) & n \in \mathbb{N} \setminus \{0\} \\ & | & \text{free}(\langle lhs \rangle) & \end{array}$$

Figure 3.1: Syntaxe des statements de $\mathcal{L}_2$

9. Couple ayant la même origine mais une destination différente.

3.2 La modélisation du tas

L'allocation dynamique implique de devoir représenter le tas pour pouvoir effectuer notre analyse. La définition de variables pour le tas est ainsi nécessaire. Ces variables seront représentées par ∗heap∗_label, où label représentera le numéro du *statement* de l'appel à malloc. Cette représentation permet de faire une analyse sensible au flot de contrôle. Une autre approche possible aurait pu être de présenter le tas sous la forme d'un tableau unique, nommé ∗heap∗. Cette dernière approche, moins précise, permet tout de même de paralléliser des boucles simples.

Le treilis *Name* de notre ensemble $\mathcal{CP}^{\#}$ est modifié en conséquence (*cf. Fig.* A.4, annexe A.4).

3.3 La routine malloc

La routine malloc permet d'allouer dynamiquement de la mémoire.

Les arguments Notre représentation de malloc prend deux arguments.
Le premier correspond à l'identifieur pour lequel on souhaite allouer de la mémoire, celui-ci doit être de type pointeur. Il nous permettra également de déterminer le type de sa représentation sur le tas.
Le deuxième argument correspond à la taille mémoire que l'on souhaite allouer. Il doit valoir soit la taille du type pointé par notre premier argument, soit être un multiple de celui-ci. Dans le dernier cas, il s'agit d'un pointeur représentant un tableau, on doit donc faire pointer vers le premier élément de celui-ci.

Génération d'arcs pour le tas On définit un ensemble Gen_{heap} (3.1) qui est chargé de créer les arcs entre l'identificateur et son emplacement abstrait sur le tas.

$$\begin{array}{rcl} Gen_{heap} : \mathcal{P}(\mathcal{CP}^{\#}) \times \mathcal{CP}^{\#} & \to & \mathcal{PT}^{\#} \\ Gen_{heap}(Source, d) & = & \{(s, d, \mathtt{MAY}), (s, \mathtt{NULL}, \mathtt{MAY})\ ^{10} | s \in Source\} \end{array} \tag{3.1}$$

Génération d'arcs *undefined* Si la mémoire allouée contient des pointeurs, il est également nécessaire de les faire pointer vers $cp_{undefined}$. On peut se définir un ensemble $Gen_{undefined}$ (3.2) qui effectuera cette vérification.

$$\begin{array}{rcl} Gen_{undefined} : \mathcal{CP}^{\#} & \to & \mathcal{PT}^{\#} \\ Gen_{undefined}(\langle var, seq, type \rangle) & = & \text{if } type = \text{pointer}(t) \\ & & \quad \{(\langle var, seq, type \rangle, cp_{undefined}), \mathtt{EXACT}\} \\ & & \text{elseif } type = \text{struct}\{id_1 : t_1, \ldots, id_n : t_n\} \\ & & \quad \bigcup\limits_{i=1,\cdots,n} Gen_{undefined}(\langle var, seq.i, t_i \rangle) \end{array} \tag{3.2}$$

Sémantique abstraite de malloc La sémantique abstraite pour la routine malloc est présentée en (3.3). κ représente le label de notre *statement*.

3.4 La routine free

La routine free permet de libérer une zone mémoire précédemment allouée par la fonction malloc.

L'argument Son argument doit donc être de type pointeur et pointer vers une partie du tas, un élément du contexte formel ou `NULL`. Dans ce dernier cas rien n'est modifié.

La routine free détruit des arcs $\mathcal{PT}^{\#}$ et génèrera des arcs vers *undefined*.

$$
\begin{array}{l}
lhs \in \langle lhs \rangle\,,\ n \in \mathbb{N} \setminus \{0\} \\
\mathbb{S}^{\#}[\![malloc(lhs, n)]\!]\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P, \mathcal{O} \rangle = \\
\quad \text{let } \langle CP_{lhs}, P_1, \mathcal{O}_1 \rangle = \mathbb{E}^{\#}_{p}[\![lhs]\!]\langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P \rangle \text{ in} \\
\quad \text{if } \forall cp_{lhs} \in CP_{lhs}, typeof(cp_{lhs}) = \text{pointeur}(type) \wedge n = m * sizeof(type) \\
\qquad \text{let } cp_{heap} = \begin{cases} newstub(\text{“}*heap*\_\kappa\text{”}, (), type) & \text{if } m == 1 \\ newstub(\text{“}*heap*\_\kappa\text{”}, (0), type)) & \text{otherwise} \end{cases} \text{ in} \\
\qquad \text{let } Gen_{heap} = Gen_{heap}(CP_{lhs}, cp_{heap}) \text{ in} \\
\qquad \text{let } Gen_{undefined} = Gen_{undefined}(cp_{heap}) \text{ in} \\
\qquad \langle \mathcal{R}^{\#}, \mathcal{S}^{\#}, P_1 \cup Gen_{heap} \cup Gen_{undefined}, \mathcal{O} \cup \mathcal{O}_1 \rangle \\
\quad \text{else } \langle \emptyset, \emptyset, \emptyset, \omega \cup \mathcal{O} \cup \mathcal{O}_1 \rangle
\end{array}
\tag{3.3}
$$

FIGURE 3.2 – Sémantique abstraite de la routine malloc

Calcul de l'ensemble *Kill* Les arcs détruits sont de deux natures : les arcs dont la source correspond à l'argument de free (3.4) ; les arcs `EXACT` dont la destination est identique aux cellules pointées par l'argument (3.5). Ce deuxième cas ne peut s'appliquer en pratique qu'à des arcs du contexte formel d'une fonction (*cf. Sec.* 1.3.3).

$$
\begin{array}{rcl}
Kill_1 : \mathcal{P}(\mathcal{CP}^{\#}) \times \mathcal{PT}^{\#} & \rightarrow & \mathcal{PT}^{\#} \\
Kill_1(Source, P) & = & \{(s, d, a) | (s, d, a) \in P \wedge s \in Source\}
\end{array}
\tag{3.4}
$$

$$
\begin{array}{rcl}
Kill_1 : \mathcal{P}(\mathcal{CP}^{\#}) \times \mathcal{PT}^{\#} & \rightarrow & \mathcal{PT}^{\#} \\
Kill_1(Dest, P) & = & \{(s, d, \texttt{EXACT}) | (s, d, \texttt{EXACT}) \in P \wedge d \in Dest
\end{array}
\tag{3.5}
$$

Calcul de l'ensemble *Gen* Comme l'ensemble $Kill$, on décompose l'ensemble Gen en deux sous-ensembles (3.6) (3.7). Ils correspondent respectivement aux deux sous-ensembles $Kill$. Pour le deuxième sous-ensemble, un nouveau calcul des approximations sera fait avec l'union $\cup^{\#}$ vue en (2.2).

$$
\begin{array}{rcl}
Gen_1 : \mathcal{P}(\mathcal{CP}^{\#}) & \rightarrow & \mathcal{PT}^{\#} \\
Gen_1(Source) & = & \text{if } Source = \{s\} \wedge atomic(s) \\
& & \quad \{(s, cp_{undefined}, \texttt{EXACT})\} \\
& & \text{else foreach } s \in Source \text{ do} \\
& & \quad \{(s, cp_{undefined}, \texttt{MAY})\}
\end{array}
\tag{3.6}
$$

$$
\begin{array}{rcl}
Gen_2 : \mathcal{P}(\mathcal{CP}^{\#}) \times \mathcal{PT}^{\#} & \rightarrow & \mathcal{PT}^{\#} \\
Gen_2(Dest, P) & = & \text{foreach } d \in Dest \text{ do} \\
& & \quad \text{if } atomic(d) \text{ then } \{(s, cp_{undefined}, \texttt{EXACT})\} \\
& & \quad \text{else } \{(s, cp_{undefined}, \texttt{MAY})\}
\end{array}
\tag{3.7}
$$

Sémantique abstraite de free On définit maintenant notre sémantique abstraite pour la routine free (3.8).

$lhs \in \langle lhs \rangle$

$$\begin{aligned}
\mathbb{S}^\#[\![free(lhs)]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P, \mathcal{O}\rangle = \ & \text{let } \langle CP_{lhs}, P_1, \mathcal{O}_1\rangle = \mathbb{E}_p^\#[\![lhs]\!]\langle \mathcal{R}^\#, \mathcal{S}^\#, P\rangle \text{ in} \\
& \text{let } CP_{target} = \{cp_t | \exists cp \in CP_{lhs}, (cp, cp_t) \in P_1 \wedge \\
& \qquad\qquad cp_t \in \{HEAP \cup FORMAL \cup anywhere \cup \texttt{NULL}\}\} \text{ in} \\
& \text{if } CP_{target} \neq \emptyset \\
& \quad \text{let } CP_{targetf} = CP_{target} \setminus \{\texttt{NULL}\} \text{ in} \\
& \quad \text{let } Kill_1 = Kill_1(CP_{lhs}, P_1) \text{ in} \\
& \quad \text{let } Kill_2 = Kill_2(CP_{targetf}, P_1) \text{ in} \\
& \quad \text{let } Gen_1 = Gen_1(CP_{lhs}) \text{ in} \\
& \quad \text{let } Gen_2 = Gen_2(CP_{targetf}, P_1) \text{ in} \\
& \quad\ \langle \mathcal{R}^\#, \mathcal{S}^\#, P_1 \setminus Kill_1 \setminus Kill_2 \cup Gen_1 \cup^\# Gen_2, \mathcal{O} \cup \mathcal{O}_1\rangle \\
& \text{else } \langle \emptyset, \emptyset, \emptyset, \omega \cup \mathcal{O} \cup \mathcal{O}_1\rangle
\end{aligned} \tag{3.8}$$

FIGURE 3.3 – Sémantique abstraite de la routine free

4 Implémentation

Dans un premier temps, je présenterai les particularités du *framework PIPS* qui diffèrent de notre formalisation. Puis, je présenterai les implémentations effectuées. La première extension consiste à utiliser l'information points-to pour raffiner les informations concernant les valeurs scalaires pointées. Cette première extension a nécessité plusieurs réflexions pour savoir quel niveau de raffinement on pouvait et on devait atteindre et avec quelles informations. La deuxième extension consiste à considérer les pointeurs en tant que variables analysables et ainsi permettre leur analyse. Ces deux extensions sont orthogonales et sont présentées en *Fig.* 4.1.

	Effect	Effect with points-to	Effect with points-to + points-to	Effect with points-to + points-to + constant path
Analyze integer	déjà présent	corrigé *Sec.* 4.2	implémenté *Sec.* 4.3	implémenté *Sec.* 4.4
Analyze integer + pointer	implémenté *Sec.* 4.5	implémenté	implémenté en partie	implémenté en partie

FIGURE 4.1 – Implémentations réalisées

4.1 Différence entre la description formelle et l'implémentation

À cause de l'architecture de *PIPS* et de certaines restrictions de celle-ci, le treillis $\mathcal{CP}^\#$ diffère quelque peu de celui présenté en *Sec.* 2.3 et détaillé en annexe A. Ces différences sont présentées en introduction de l'annexe A lors de la comparaison avec le treillis de Mensi.

De plus, *PIPS* lance des passes d'analyses ou de transformations successives. Ainsi, pour réaliser une analyse sémantique qui correspond aux passes *Transformers* ou *Preconditions*, on a besoin d'une autre passe qui nous permet de calculer les effets sur les variables. Cette dernière s'appelle *Effect* peut être générée avec ou sans informations de la passe *points-to*. Le travail effectué se situe notamment sur les passes *Transformers* pour y rajouter les extensions demandées.

Durant la passe *Points-to*, la prise d'adresse d'une variable ($\&i$) ne crée pas l'arc entre l'adresse et la variable elle-même ($\&i \to i$). Néanmoins, cet arc devra peut-être être rajouté si l'on souhaite mettre à jour l'ensemble $\mathcal{PT}^\#$ à partir des nouvelles informations de notre passe *Transformers*.

4.2 Correction de l'analyse avec *Effect with points-to*

L'analyse *points-to* dans *PIPS* étant récemment implémentée, certains problèmes sont apparus. Ainsi, j'ai pu remarquer que la passe calculant les *transformers* à partir de *Effect with points-to* ne mettait pas à jour les informations présentes.

En effet, on détectait bien les variables à modifier, mais on ne les modifiait pas. Le choix fait pour cette correction est de perdre les informations concernant les variables modifiées. Pour avoir les nouvelles valeurs de la variable, le graphe $\mathcal{PT}^{\#}$ est nécessaire *Sec.* 4.3. Ce choix permet de normaliser le calcul des *transformers* en présence uniquement des effets, ces derniers calculés avec ou sans *points-to*. En effet, dans un premier temps, on avait décidé de donner la ou les nouvelles valeurs prises lorsque c'était possible, avant avant d'opter pour une différenciation claire des résultats obtenus avec juste *Effect with points-to* et avec la présence du graphe $\mathcal{PT}^{\#}$.

Les résultats de cette correction d'analyse sont présentés en annexe D.

4.3 Implémentation de l'utilisation du graphe $\mathcal{PT}^{\#}$

L'implémentation de l'analyse utilisant le graphe $\mathcal{PT}^{\#}$ a nécessité la création d'une nouvelle passe dans *PIPS*. En effet, cette analyse, comme son nom l'indique, en plus du prérequis de connaître les effets, nécessite d'avoir le graphe $\mathcal{PT}^{\#}$ également connu.

Ainsi, lors de cette analyse, lorsque l'on rencontre une variable de type pointeur, on se réfère au graphe $\mathcal{PT}^{\#}$ pour savoir sur quelles valeurs ou autres variables, le pointeur peut pointer. S'il peut pointer sur différentes valeurs ou variables, on calcule l'enveloppe convexe de ces destinations. Cette dernière correspond aux résultats obtenus. De plus, si ce pointeur se situe en partie droite d'une affectation, on renvoie une erreur si le pointeur pointe uniquement vers *undefined* ou vers `NULL`, et une alerte s'il peut pointer vers une de ces deux valeurs, exception faite pour l'affectation de la valeur `NULL` à un pointeur de pointeur.

Les résultats de cette implémentation sont présentés en annexe E.

4.4 Implémentation de l'analyse avec $\mathcal{CP}^{\#}$

Dans un premier temps, on avait envisagé et souhaité que cette analyse soit également orthogonale aux deux autres extensions *Sec.* 4.3 et *Sec.* 4.5. Mais l'architecture de *PIPS* ne le permettait pas ou l'aurait rendu incohérente.

En effet, avec l'architecture et les fonctions actuellement présentes dans *PIPS*, le moyen le plus sûr d'avoir un chemin d'accès constant valide est de le récupérer à partir du graphe $\mathcal{PT}^{\#}$ avec la fonction *expression_to_points_to_sources* pour un élément gauche d'une affectation et avec la fonction *expression_to_points_to_sinks* pour un élément droit d'une affectation. Ainsi, il serait incohérent de faire une nouvelle passe ou une nouvelle propriété indépendante de la passe précédente. De plus, le travail nécessaire pour permettre d'obtenir un chemin constant uniquement avec une expression était trop coûteux dans le temps imparti pour le stage. En effet, il aurait fallu, en plus de générer ce chemin, être sûr qu'il correspondait bien au même chemin que celui que pouvait générer le calcul du graphe $\mathcal{PT}^{\#}$ au risque de ne pas pouvoir utiliser ce dernier. Étant donné que les chemins constants servent pour la résolution de pointeurs en tant que paramètre formel, il n'est pas déraisonnable de considérer que le traitement des chemins d'accès constant comme uniquement une extension de l'utilisation du graphe $\mathcal{PT}^{\#}$, *Sec.* 4.3.

Ce traitement des chemins constants permet également de pouvoir faire certaines analyses sur les structures qui n'étaient pas possibles avant. En effet, les éléments d'une structure sont considérées comme étant un chemin constant. Il permet également d'analyser les valeurs des éléments d'un tableau.

Pour utiliser cette analyse, il faut activer la propriété *SEMANTICS_ANALYZE_CONSTANT_PATH* dans *PIPS*.

Les résultats de cette implémentation sont présentés en annexe F.

4.5 Implémentation de l'analyse des pointeurs

L'analyse des pointeurs est une extension orthogonale à l'analyse utilisant le graphe $\mathcal{PT}^{\#}$ présentée précédemment.

Il faut activer la propriété *SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES*, pour réaliser cette analyse dans *PIPS*.

La réalisation de cette analyse s'est déroulée en deux temps. Dans un premier temps, il a fallu ajouter la signification d'un référencement. Dans un second temps, on a considéré l'arithmétique concernant les pointeurs.

Une étape préliminaire a également été effectuée pour mettre à jour le filtre pour détecter la présence et l'analyse des pointeurs.

Le référencement Le traitement du référencement ne doit être considéré qu'en partie droite d'une affectation, $p = \&i$.

Plusieurs solutions ont été envisagées pour cette implémentation :

1. Transformer le code à analyser en ajoutant des variables de transition.
   Par exemple, $p = \&i;$ deviendrait $\_p\_i = \&i; p = \_p\_i;$.
2. Ajouter une représentation propre dans la représentation interne de *PIPS*.
3. Interpréter &i par une valeur abstraite durant l'analyse.

La solution 1 a pour défaut de modifier le code, or dans *PIPS* on ne souhaite pas modifier le code à moins que ce soit explicitement demandé par l'utilisateur durant une passe de transformation. De plus, les modifications de code se font en vue d'optimisation du code. La solution 2 a le risque de causer plusieurs effets de bord sur l'analyse déjà présente dans *PIPS*. La solution 3 permet de représenter &i dans *PIPS* durant l'analyse, sans toutefois modifier la représentation interne. C'est pourquoi les deux premières solutions n'ont pas été retenues et que la dernière a été implémentée.

Les résultats de cette implémentation sont présentés en annexe G.

L'arithmétique Dans un premier temps, on a repris l'arithmétique concernant les entiers pour faire cette arithmétique sur les pointeurs. Il faut savoir que *PIPS* refuse d'analyser du code qui ne passe pas une compilation *gcc*. Ainsi, on n'a pas effectué des vérifications telles que l'ajout de deux pointeurs, la multiplications de pointeurs, *etc.*

Cette solution n'est néanmoins pas satisfaisante car les pointeurs doivent être typés. Ainsi, la valeur 1 n'a pas la même signification dans $i = i + 1$, $p = p + 1$ ou $q = q + 1$ lorsque i est un entier et p et q sont des pointeurs de types différents.

Plusieurs solutions peuvent être envisagées dans ce cas également :

1. Effectuer un typage des contraintes, donc ne pas faire d'enveloppe convexe, ou de fermeture transitive sur des variables de types différents mais les dissocier.
2. Multiplier la valeur typée par le *sizeof* de son type, par exemple $p = p + sizeof(*p)$.
3. Restreindre l'arithmétique des pointeurs sur des pointeurs représentant des tableaux, et ainsi se déplacer dans les cases du tableaux.

La solution 1 semble difficilement implémentable avec l'architecture déjà très avancée de *PIPS* et tous les effets de bord que cela pourrait causer. La solution 2 est plus générale que la solution 3, et semble régler les problèmes de typages. La solution 3 bien que plus contraignante pourrait bien être plus exacte et détecter des mauvaises pratiques de codage. En effet, l'arithmétique sur les pointeurs doit normalement servir à se déplacer dans les cases d'un tableaux[ISO07]. Si l'on fait $p = p + 1$ alors que p n'est pas un tableau, on arrive dans une adresse mémoire dont on ne sait rien à priori et la modifier pourrait avoir des répercussions inconnues.

L'implémentation réalisée est celle de la solution 2. J'aurais néanmoins voulu également pouvoir implémenter la solution 3 pour pouvoir effectuer des comparaisons et voir laquelle offrait les meilleurs résultats. La solution 3 a pour défaut que *PIPS* ne représentait pas ou difficilement les valeurs d'un tableau ; l'extension *Sec.* 4.4 permet maintenant de le faire en partie. Mais le temps m'a manqué pour réaliser cette implémentation et ces tests.

Les résultats de cette implémentation sont présentés en annexe G.

5 Conclusion

Durant ce stage, j'ai pu découvrir le fonctionnement d'un analyseur et transformateur de code source, *PIPS*.

De plus, j'ai pu formaliser la sémantique d'une analyse sur les pointeurs, à partir de travaux antérieurs. J'ai également pu vérifier que cette description était valable par mon implémentation et les résultats obtenus grâce à celle-ci.

Ainsi, j'ai pu élargir l'ensemble des applications qui peuvent être modélisées et donc analysées et transformées à l'aide d'outils automatiques ou semi-automatiques.

Des travaux restent tout de même à faire pour pouvoir traiter tous les types et possibilités qu'offre le langage `C` pour les pointeurs, à savoir la possibilité de traiter l'union et de faire des *cast*. L'analyse utilisant $\mathcal{CP}^{\#}$ nécessite encore quelques corrections. L'analyse inter-procédurale est également en cours d'implémentation, mais nécessite également une formalisation de sa sémantique. De même, l'implémentation de l'arithmétique sur les pointeurs serait peut être à revoir pour envisager la troisième solution proposée en *Sec.* 4.5. Enfin, un autre travail à faire serait d'étudier la possibilité de raffiner le graphe $\mathcal{PT}^{\#}$ à partir de notre analyse sémantique et de déterminer quand ces raffinements entre l'analyse sémantique et l'analyse *points-to* doivent s'arrêter.

Annexes

A Les treillis composants $\mathcal{CP}^{\#}$

Cette annexe donne la description des treillis composants $\mathcal{CP}^{\#}$, notamment la description des relations $kill_{\texttt{MAY}}$ et $kill_{\texttt{EXACT}}$ des différents sous-treillis.

Des différences sont à noter par rapport aux treillis décrits dans la thèse de Mensi [Men13].

Je n'ai pas le sous-treillis *Module* car je ne parle pas de l'aspect inter-procédural. Mais ce treillis, s'il devait être introduit, serait le même que celui de Mensi et reste un treillis très simple. Chaque nom de module correspond à un nom de fonction, et l'on rajoute un plus petit et un plus grand élément pour assurer la complétude du treillis.

La grande différence repose dans la description des treillis *Name* et V_{Ref}. En effet, lorsque je forme le contexte formel à la demande ou que j'accède à un champ d'une structure, le nom présent dans *Name* ne change pas, mais je rajoute un indice dans V_{Ref}. Ce choix a été fait pour normaliser la description de notre sémantique. Il a pour impact de donner beaucoup plus d'importance au treillis V_{Ref} et de rendre indispensable les informations du treillis *Type* pour l'analyse. De ce fait, j'ai pu remarquer quelques erreurs dans la description de ces treillis chez Mensi et les ai corrigées. En effet, Mensi n'apporte que peu d'importance au treillis *Type* car un gros travail a été fourni lors de la génération de nouveaux noms dans le treillis *Name* qui permettent très souvent de se passer des informations du treillis *Type*. Ces deux descriptions de treillis restent équivalents.

Lors de l'implémentation, c'est le treillis de Mensi qui est utilisé. L'une des raison de ce choix est une contrainte d'unicité sur les noms des objets due à *PIPS*.

A.1 Treillis *Name* de $\mathcal{L}_1$

Pour assurer la complétude du treillis $Name$, nous rajoutons un plus petit et un plus grand élément, $\perp$ et $\top$. De plus, nous rajoutons également l'élément $defined$, ou $anywhere$, pour s'opposer à *undefined*. Les variables sont également différenciées selon le fait qu'elles sont en paramètre formel, `FORMAL`, ou qu'elles sont déclarées de façon statique, `STATIC`. On obtient ainsi le treillis *Fig.* A.1.

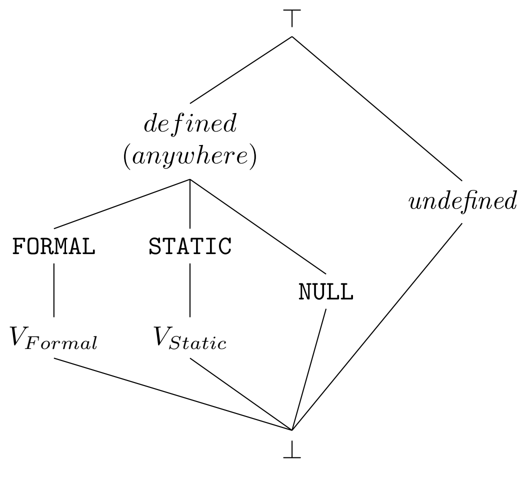


FIGURE A.1 – Treillis $Name$ $\mathcal{L}_1$

Les opérateurs $kill_{\texttt{MAY}}$ (A.1) et $kill_{\texttt{EXACT}}$ (A.2) sont assez similaires à $\subseteq^{\#}$. Mais ils peuvent renvoyer ω qui correspond à une erreur de localisation.

$$\begin{aligned} name_1 kill_{\texttt{MAY}} name_2 &= \text{if } name_2 \in \{\perp, \texttt{NULL}, undefined\} \text{ then } \omega \\ &\quad \text{else } name_2 \subseteq^{\#} name_1 \end{aligned} \tag{A.1}$$

$$\begin{aligned} name_1 kill_{\texttt{EXACT}} name_2 &= \text{if } name_2 \in \{\perp, \texttt{NULL}, undefined\} \text{ then } \omega \\ &\quad \text{elseif } name_2 = \top \text{ then } false \\ &\quad \text{else } name_2 \subseteq^{\#} name_1 \end{aligned} \tag{A.2}$$

A.2 Treillis V_{Ref} de $\mathcal{L}_1$

V_{Ref} correspond en réalité à un ensemble de treillis et non pas à un treillis unique. En effet, il dépendra de la taille de la séquence que l'on considère. L'élément minimal de tous ces treillis sera la séquence de 0 éléments () et l'élément maximal sera la séquence composée uniquement de $*$.

Le treillis de zéro élément est un treillis dégénéré où $\top = \perp$. Le treillis de séquence d'un élément est donné à la figure *Fig.* A.2a, on l'appellera $T_{V_{Ref},1}$. On peut créer les treillis suivants de manière récursive. Je prendrai le cas du treillis de profondeur $n = 2$, *Fig.* A.2b, pour expliquer cette récursion. On part du treillis de profondeur $n-1$, c'est à dire de profondeur 1 dans notre exemple, en vert sur la figure. Sur chaque élément du treillis, on applique le treillis $T_{V_{Ref},1}$, en noir. On met également en relation les séquences de longueur n possédant une valeur $v \in \mathbb{N}$ avec la même séquence mais avec la valeur $v = *$, en rouge et en bleu. On obtient ainsi notre treillis $T_{V_{Ref},2}$, *Fig.* A.2b. On a ainsi la définition suivante :

$$(v_1, \dots, v_n, v_{n+1}) \subseteq^{\#} (v'_1, \dots, v'_n, v'_{n+1}) \text{ iff } (v_1, \dots, v_n) \subseteq^{\#} (v'_1, \dots, v'_n) \wedge v_{n+1} \subseteq^{\#} v'_{n+1} \tag{A.3}$$

Notre treillis V_{Ref} est ainsi défini par : $T_{V_{Ref}} = \bigcup\limits_{n \in \mathbb{N}} T_{V_{Ref},n}$

On considère qu'une séquence s est équivalente à la séquence $s.\perp$, c'est-à-dire que l'on peut concaténer $\perp$ à une séquence sans la modifier. De plus, s_1 correspondra au premier élément de la séquence s et s_{rest} à la séquence s privée de son premier élément, c'est-à-dire commençant au deuxième élément. Ces définitions permettent de spécifier nos relations $kill_{\texttt{MAY}}$ (A.4) et $kill_{\texttt{EXACT}}$ (A.5).

$$\begin{aligned} s\ kill_{\texttt{MAY}}\ s' &= \text{if } s_1 = \perp \text{ then } true \\ &\quad \text{elseif } s_1 = s'_1 \vee s_1 = * \vee s'_1 = * \text{ then } s_{rest}\ kill_{\texttt{MAY}}\ s'_{rest} \\ &\quad \text{else } false \end{aligned} \tag{A.4}$$

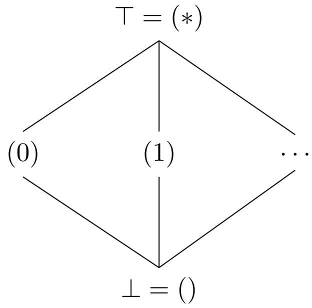
$$\begin{aligned} s\ kill_{\texttt{EXACT}}\ s' &= \text{if } s_1 = \perp \text{ then } true \\ &\quad \text{elseif } s_1 = s'_1 \text{ then } s_{rest}\ kill_{\texttt{EXACT}}\ s'_{rest} \\ &\quad \text{else } false \end{aligned} \tag{A.5}$$

A.3 Treillis *Type* de $\mathcal{L}_1$

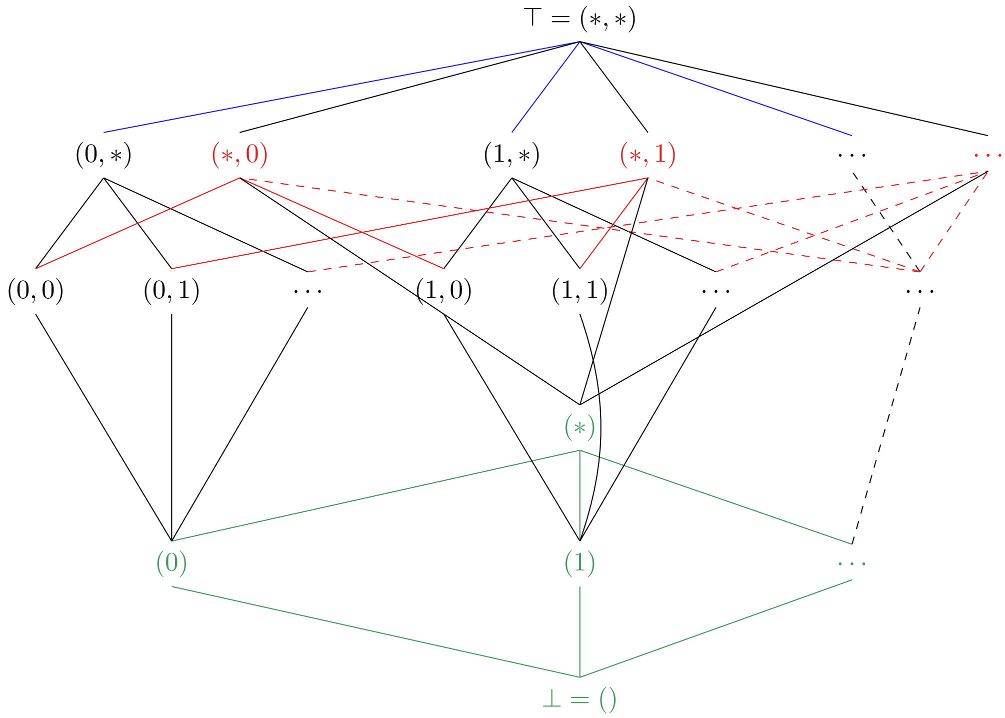
Le treillis *Type* est présenté ci-dessous *Fig.* A.3. L'élément maximal correspond à *overloaded* et le minimal à *type_unknown*.

On doit également considérer la relation d'équivalence [11] des types que l'on définit par l'opérateur $\sim$ (A.6). Comme le treillis *Name*, les opérateurs $kill_{\texttt{MAY}}$ (A.7) et $kill_{\texttt{EXACT}}$ (A.8)

11. Une relation d'équivalence est une relation réflexive, transitive et symétrique.



(a) Treillis V_{Ref} de profondeur 1



(b) Treillis V_{Ref} de profondeur 2

FIGURE A.2 – Treillis V_{Ref}

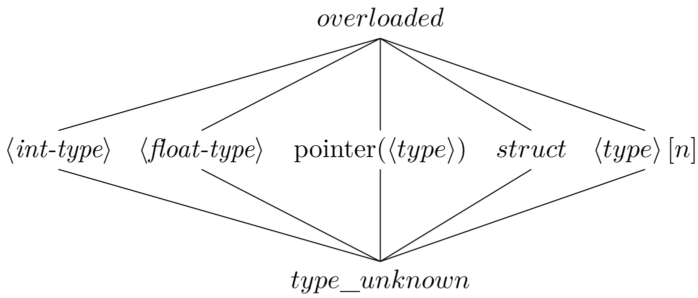


FIGURE A.3 – Treillis *Type*

sont assez similaires à $\subseteq^{\#}$ auquel on rajoute notre relation d'équivalence.

$$
\begin{aligned}
type_1 \sim type_2 \quad &= \quad \text{if } type_1 = type_2 \text{ then } true && \text{(A.6)}\\
&\quad\quad \text{elseif } (type_1 = \text{pointeur}(type_{1p}) \wedge \; type_2 = \text{pointeur}(type_{2p}[n]))\\
&\quad\quad\quad\quad \vee \; (type_1 = \text{pointeur}(type_{1p}[n]) \wedge \; type_2 = \text{pointeur}(type_{2p}))\\
&\quad\quad\quad type_{1p} \sim type_{2p}\\
&\quad\quad \text{else } false\\
type_1 kill_{\mathtt{MAY}} type_2 \quad &= \quad \text{if } type_2 = overloaded \text{ then } true && \text{(A.7)}\\
&\quad\quad \text{else } type_2 \subseteq^{\#} type_1 \vee type_1 \sim type_2\\
type_1 kill_{\mathtt{EXACT}} type_2 \quad &= \quad \text{if } type_1 = overloaded \text{ then } false && \text{(A.8)}\\
&\quad\quad \text{else } type_2 \subseteq^{\#} type_1 \vee type_1 \sim type_2
\end{aligned}
$$

A.4 Treillis *Name* de $\mathcal{L}_2$

Le langage $\mathcal{L}_2$ introduit l'allocation dynamique et la modélisation du tas est nécessaire.

Le treillis *Name* de notre ensemble $\mathcal{CP}^{\#}$ est modifié en conséquence pour prendre en compte cette nouvelle localisation, *Fig.* A.4. Les opérations sur ce treillis restent inchangées.

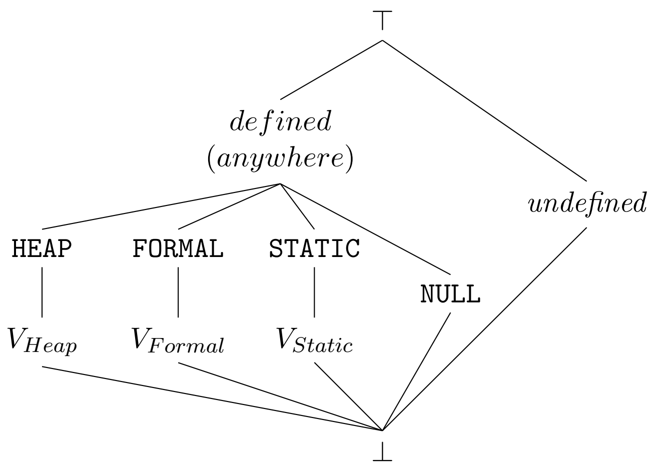


FIGURE A.4 – Treillis *Name*

B Analyse détaillée de l'exemple pour le langage $\mathcal{L}_0$

```
struct bar {int **pp1; int **pp2;};

void foo(struct bar *ps) {
  int i = 1, j = 2, *p;       //S1
  p = &j;                     //S2
  *((*ps).pp1) = &i;          //S3
  *((*ps).pp2) = p;           //S4
  (*ps).pp1 = (*ps).pp2;      //S5
}
```

CODE 4 – Exemple pour la sémantique du langage $\mathcal{L}_0$

L'analyse de ce code dans notre sémantique abstraite donnera les résultats suivants.

Le *statement* S1 initialise notre environnement de variable et de pointeur. On aura :

$$\begin{array}{ll}
\rho(i) = cp_i = \langle i, (), int \rangle & \sigma(cp_i, \emptyset) = 1 \\
\rho(j) = cp_j = \langle j, (), int \rangle & \sigma(cp_j, \emptyset) = 2 \\
\rho(p) = cp_p = \langle p, (), \text{pointer}(int) \rangle & \\
cp_{undefined} = & P^{S1} = \{(cp_p, cp_{undefined})\} \\
\quad \langle undifined, (), overloaded \rangle &
\end{array}$$

Le *statement* S2 permet de faire pointer p vers j. Dans un premier temps, on détermine la sémantique pour p et $\&j$. Puis, on calcul les ensembles $Kill$ et Gen. Enfin, on donnera la sémantique de l'affectation.

$$\dfrac{\dfrac{\dfrac{\begin{array}{ll} & \mathbb{E}_p^{\#}[\![p]\!]\langle\{\rho\},\{\sigma\},P^{S1}\rangle = \langle cp_p, P^{S1}, \emptyset\rangle \end{array}}{\begin{array}{ll} \rho(\&j) = cp_{\&j} = & \mathbb{E}_p^{\#}[\![\&j]\!]\langle\{\rho\},\{\sigma\},P^{S1}\rangle = \\ \quad \langle j\#address, \emptyset, \text{pointer}(int)\rangle & \quad \langle cp_{\&j}, P^{S1} \cup^{\#} \{(cp_{\&j}, cp_j)\}, \emptyset\rangle \end{array}}}{\begin{array}{c} Kill^{S2} = \{(cp_p, cp_{undefined})\} \\ Gen^{S2} = \{(cp_p, cp_j)\} \end{array}}}{\dfrac{\begin{array}{ll} \sigma(cp_p, P^{S2}) = \sigma(cp_{\&j}, P^{S2}) & \mathbb{S}^{\#}[\![p = \&j]\!]\langle\{\rho\},\{\sigma\},P^{S1},\emptyset\rangle = \\ & \quad \langle\{\rho\},\{\sigma\},\{(cp_{\&j}, cp_j),(cp_p, cp_j)\},\emptyset\rangle \end{array}}{P^{S2} = \{(cp_{\&j}, cp_j),(cp_p, cp_j)\}}}$$

Le *statement* S3 initialise le premier champ de la structure. Comme précédemment, on donne la sémantique des expressions, les ensembles $Kill$ et Gen et la sémantique de l'affectation.

$$\dfrac{\dfrac{\dfrac{\begin{array}{ll} \rho(ps) = cp_{ps} = & \\ \quad \langle ps, (), (\text{struct}\{int**, int**\})*\rangle & \mathbb{E}_p^{\#}[\![*((*ps).pp1)]\!]\langle\{\rho\},\{\sigma\},P^{S2}\rangle = \\ \rho(*ps) = cp_{ps[0]} = & \quad \langle cp_{ps[0].pp1[0]}, P_1^{S2}, \emptyset\rangle \\ \quad \langle ps, (0), \text{struct}\{int**, int**\}\rangle & \\ cp_{ps[0].pp1} = & \quad P_1^{S2} = P^{S2} \cup^{\#} \{ \\ \quad \langle ps, (0;1), \text{pointer}(int*)\}\rangle & \qquad (cp_{ps}, cp_{ps[0]}), \\ \rho(*(*ps.pp1) = cp_{ps[0].pp1[0]} = & \qquad (cp_{ps[0].pp1}, cp_{ps[0].pp1[0]})\} \\ \quad \langle ps, (0;1;0), \text{pointer}(int)\}\rangle & \end{array}}{\begin{array}{ll} \rho(\&i) = cp_{\&i} = & \mathbb{E}_p^{\#}[\![\&i]\!]\langle\{\rho\},\{\sigma\},P_1^{S2}\rangle = \\ \quad \langle i\#address, \emptyset, \text{pointer}(int)\rangle & \quad \langle cp_{\&i}, P_1^{S2} \cup^{\#} \{(cp_{\&i}, cp_i)\}, \emptyset\rangle \end{array}}}{\begin{array}{c} Kill^{S3} = \emptyset \\ Gen^{S3} = \{(cp_{ps[0].pp1[0]}, cp_i)\} \end{array}}}{\dfrac{\begin{array}{ll} \sigma(cp_{ps[0].pp1[0]}, P^{S3}) = \sigma(cp_{\&i}, P^{S3}) & \mathbb{S}^{\#}[\![*((*ps).pp1) = \&i]\!]\langle\{\rho\},\{\sigma\},P^{S2},\emptyset\rangle = \\ & \quad \langle\{\rho\},\{\sigma\},P^{S3},\emptyset\rangle \end{array}}{P^{S3} = P_1^{S2} \cup^{\#} \{(cp_{\&i}, cp_i),(cp_{ps[0].pp1[0]}, cp_i)\}}}$$

De même pour le *statement* S4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{ll} cp_{ps[0].pp2} = & \\ \quad \langle ps,(0;2),\text{pointer}(int*)\}\rangle & \mathbb{E}_p^\#[\![\, *((*ps).pp2)]\!]\langle\{\rho\},\{\sigma\},P^{S3}\rangle = \langle cp_{ps[0].pp2[0]}, P_1^{S3},\emptyset\rangle \\ \rho(*(*ps.pp2) = cp_{ps[0].pp2[0]} = & \\ \quad \langle ps,(0;2;0),\text{pointer}(int)\}\rangle & P_1^{S3} = P^{S3} \cup^\# \{(cp_{ps[0].pp2}, cp_{ps[0].pp2[0]})\} \end{array}}{\mathbb{E}_p^\#[\![p]\!]\langle\{\rho\},\{\sigma\},P_1^{S3}\rangle = \langle cp_p, P_1^{S3},\emptyset\rangle}}{\begin{array}{c} Kill^{S4} = \emptyset \\ Gen^{S4} = \{(cp_{ps[0].pp2[0]}, cp_j)\} \end{array}}}{\begin{array}{ll} \sigma(cp_{ps[0].pp2[0]}, P^{S4}) & \\ \quad = \sigma(cp_p, P^{S4}) & \mathbb{S}^\#[\![\, *((*ps).pp2) = p]\!]\langle\{\rho\},\{\sigma\},P^{S3},\emptyset\rangle = \langle\{\rho\},\{\sigma\},P^{S4},\emptyset\rangle \\ \quad = \sigma(cp_{\&i}, P^{S4}) & \end{array}}}{P^{S4} = P_1^{S3} \cup^\# \{(cp_{ps[0].pp2[0]}, cp_j)\}}$$

Enfin, le *statement* S5 nous donne.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{E}_p^\#[\![(*ps).pp1]\!]\langle\{\rho\},\{\sigma\},P^{S4}\rangle = \langle cp_{ps[0].pp1}, P^{S4},\emptyset\rangle}{\mathbb{E}_p^\#[\![(*ps).pp2]\!]\langle\{\rho\},\{\sigma\},P^{S4}\rangle = \langle cp_{ps[0].pp2}, P^{S4},\emptyset\rangle}}{\begin{array}{c} Kill^{S5} = \{(cp_{ps[0].pp1}, cp_{ps[0].pp1[0]})\} \\ Gen^{S5} = \{(cp_{ps[0].pp1}, cp_{ps[0].pp2[0]})\} \end{array}}}{\begin{array}{ll} \sigma(cp_{ps[0].pp1}, P^{S5}) & \mathbb{S}^\#[\![(*ps).pp1 = (*ps).pp2]\!]\langle\{\rho\},\{\sigma\},P^{S4},\emptyset\rangle = \langle\{\rho\},\{\sigma\},P^{S5},\emptyset\rangle \\ \quad = \sigma(cp_{ps[0].pp2}, P^{S5}) & \end{array}}}{\begin{array}{c} P^{S5} = \{(cp_{\&j}, cp_j), (cp_p, cp_j), (cp_{ps}, cp_{ps[0]}), (cp_{\&i}, cp_i), (cp_{ps[0].pp1[0]}, cp_i), \\ (cp_{ps[0].pp2}, cp_{ps[0].pp2[0]}), (cp_{ps[0].pp2[0]}, cp_j), (cp_{ps[0].pp1}, cp_{ps[0].pp2[0]})\} \end{array}}$$

À la fin de notre analyse, nous obtenons les propriétés suivantes :

$$P = \{ \quad \begin{array}{ccr} & & (cp_{\&i}, cp_i), \\ & & (cp_{ps[0].pp1[0]}, cp_i), \\ (cp_{ps}, cp_{ps[0]}), & \begin{array}{l} (cp_{ps[0].pp1}, cp_{ps[0].pp2[0]}), \\ (cp_{ps[0].pp2}, cp_{ps[0].pp2[0]}), \end{array} & (cp_{ps[0].pp2[0]}, cp_j), \\ & & (cp_{\&j}, cp_j), \\ & & (cp_p, cp_j)\} \end{array}$$

$$\begin{array}{lll} \sigma(cp_i, P) & = 1 & \\ \sigma(cp_j, P) & = 2 & \\ \sigma(cp_{ps[0].pp1[0]}, P) & = \sigma(cp_{\&i}, P) & \\ \sigma(cp_{ps[0].pp2[0]}, P) & = \sigma(cp_p, P) & = \sigma(cp_{\&j}, P) \\ \sigma(cp_{ps[0].pp1}, P) & = \sigma(cp_{ps[0].pp2}, P) & \end{array}$$

C Description et fonctionnement de *PIPS* [PIP]

Cette annexe présente une description très sommaire de *PIPS* et de son fonctionnement. Cette description est présentée pour aider à comprendre les résultats obtenus dans les annexes de résultats D, E, F, G. Pour plus de détails concernant *PIPS*, il faut se référer au site `http://www.pips4u.org/` [PIP].

PIPS est un *framework* de compilation de code source-à-source pour l'analyse et la transformation de code `C` et Fortran.

Le *workspace* *PIPS* a besoin d'un *workspace* pour pouvoir s'exécuter. Ce *workspace* correspond au fichiers ou à l'ensemble des fichier à traiter. Il permettra entre autre de stocker les ressources disponibles pour les traitements à faire.

Les règles et propriétés L'exécution d'une analyse ou d'une transformation s'effectuent au moyen de règles ou d'ensembles de règles, appelé passe ou phase. Par abus, on utilisera indifféremment ces trois termes.

Plusieurs types de règles existent dans *PIPS*, elles sont principalement réunies en deux groupes : celles qui permettent de réaliser une analyse de code en générant des ressources dans le *workspace* ; et celles qui permettent de réaliser une transformation directement dans le code.

Nous n'utiliserons que certaines de ces passes, à savoir l'analyse des effets, l'analyse des *points-to*, l'analyse des *transformers* et les *pretty-printers* correspondants. Les *pretty-printers* sont des passes spéciales qui permettent de rajouter en commentaires les résultats obtenus par une analyse dans le code analysé.

Ces passes sont paramétrables au moyen de propriétés.

L'interface *tpips* On utilise l'interface *tpips* permettant d'exécuter un traitement au moyen de lignes de commandes. Il permet également d'écrire des scripts. Dans *tpips*, la gestion du *workspace* se fait par les commandes *`create`*, *`open`*, *`close`* ou *`delete`*. L'utilisation explicite d'une règle plutôt qu'une autre nécessite la commande *`activate`*. Enfin la configuration d'une propriété demande la commande *`setproperty`*. Des règles et des propriétés par défaut sont déjà présentes lors de l'utilisation de *tpips*.

Un script *tpips* se compose principalement de quatre parties. Au début, la création du *workspace* est faite. Puis, on configure les propriétés voulues. Ensuite, l'analyse et les transformations sont effectuées. Enfin, on ferme et détruit, si nécessaire, le *workspace*.

D Résultat de la correction d'analyse *Effect with points-to*

Cette correction permet de supprimer les informations concernant les variables pointées lors de l'affectation par déréférencement avec *Effect with points-to*. Avant cette correction, bien que les variables modifiées étaient détectées, leur valeur n'était pas supprimée. Sans *Effect with points-to*, étant donné que l'on ne sait pas ce qui est modifié, une approche conservative est adoptée, c'est-à-dire que toutes les valeurs des variables sont supprimées.

Cette analyse nécessite la passe *PROPER_EFFECTS_WITH_POINTS_TO*. Notre exemple utilisera les script *tpips* *Fig.* D.1.

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE

echo
echo // PROPER EFFECTS
activate PRINT_CODE_PROPER_EFFECTS
display PRINTED_FILE[main]

echo
echo // Transformers
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(a) Script sans *Effect with points-to*

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE

echo
echo // PROPER EFFECTS with point-to
activate PROPER_EFFECTS_WITH_POINTS_TO
activate PRINT_CODE_PROPER_EFFECTS
display PRINTED_FILE[main]

echo
echo // Transformers with PROPER EFFECTS with point-to
activate PROPER_EFFECTS_WITH_POINTS_TO
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(b) Script avec *Effect with points-to*

FIGURE D.1 – Scripts *tpips* pour comparer l'analyse avec et sans *Effect with points-to*

L'exemple *Fig.* D.2 effectue une affectation au moyen d'un déréférencement avec un pointeur. Ainsi, on initialise dans un premier temps les variables `i` et `j` à `1`. Puis, on fait pointer `p` sur `i`. Enfin, par déréférencement sur `p`, on souhaite modifier la variable `i`.

Les résultats sans *Effect with points-to* D.2b ne permettent de rien dire lorsque l'affectation est effectuée. Ainsi, on perd toutes les valeurs de toutes les variables.

L'analyse avec *Effect with points-to* avant la correction D.2c détectait bien que la variable `i` était modifiée. Néanmoins, l'analyse ne modifiait ou ne supprimait pas la valeur de `i`. Ainsi, `i` valait toujours `1` pour l'analyse ce qui était faux.

D.2d présente les résultats après la correction de l'analyse. Ainsi, maintenant, la valeur d'une variable modifiée par déréférencement est supprimée avec uniquement l'information fournie par *Effect with points-to*, c'est-à-dire dans notre cas que l'on ne dispose plus d'information sur `i`. Ce choix a été fait pour rester cohérent avec les résultats que donnent les analyses se basant sur les effets pour modifier la valeur pointée par des pointeurs. Pour avoir la ou les nouvelles valeurs possibles, il faut utiliser le graphe $\mathcal{PT}^\#$ dont les résultats sont présentés dans l'annexe suivante *Sec.* E.

```
int main() {
  int i=1, j=1;
  int *p;

  p=&i;
  *p=0;
  return *p;
}
```

(a) Code à analyser

```
// PROPER EFFECTS
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;

//  <    is written>: p
   p = &i;
//  <may be written>:
     *ANY_MODULE*:*ANYWHERE*
//  <    is read   >: p
   *p = 0;
//  <may be read  >:
     *ANY_MODULE*:*ANYWHERE*
   return *p;
}

// Transformers
//  T(main) {}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;

//  T(i,j) {i#init==1, j#init==1}
   *p = 0;

//  T(main) {}
   return *p;
}
```

(b) Résultat sans *Effect with points-to*

```
// PROPER EFFECTS with point-to
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;
//  <    is written>: p

   p = &i;
//  <    is read   >: p
//  <    is written>: i
   *p = 0;
//  <    is read   >: i p
   return *p;
}

// Transformers with PROPER
     EFFECTS with point-to
//  T(main) {}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i==1, i#init==1, j==1}
   *p = 0;

//  T(main) {i==1, j==1}
   return *p;
}
```

(c) Résultat avant correction

```
// PROPER EFFECTS with point-to
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;

//  <    is written>: p
   p = &i;
//  <    is read   >: p
//  <    is written>: i
   *p = 0;
//  <    is read   >: i p
   return *p;
}

// Transformers with PROPER
     EFFECTS with point-to
//  T(main) {}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i#init==1, j==1}
   *p = 0;

//  T(main) {j==1}
   return *p;
}
```

(d) Résultat après correction

FIGURE D.2 – Code avec déréférencement

E Résultat de l'implémentation utilisant le graphe $\mathcal{PT}^{\#}$

L'utilisation du graphe $\mathcal{PT}^{\#}$ permet d'affecter les nouvelles valeurs d'une variable modifiée en présence de déréférencement.

Cette analyse nécessite la passe *TRANSFORMERS_INTER_FULL_WITH_POINTS_TO*. Nos exemples utiliseront les scripts *tpips* *Fig.* E.1.

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE

activate PROPER_EFFECTS_WITH_POINTS_TO
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(a) Script sans $\mathcal{PT}^{\#}$

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE

activate PROPER_EFFECTS_WITH_POINTS_TO
activate TRANSFORMERS_INTER_FULL_WITH_POINTS_TO
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(b) Script avec $\mathcal{PT}^{\#}$

FIGURE E.1 – Scripts *tpips* pour comparer l'analyse avec et sans $\mathcal{PT}^{\#}$

Le premier exemple *Fig.* E.2 correspond au même code à analyser que celui *Fig.* D.2 *Sec.* D. Mais l'utilisation du graph $\mathcal{PT}^{\#}$ permet de donner la nouvelle valeur à `i` à savoir `0`.

```c
int main() {
  int i=1, j=1;
  int *p;

  p=&i;
  *p=0;
  return *p;
}
```

(a) Code à analyser

```c
//  T(main) {}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i#init==1, j==1}
   *p = 0;

//  T(main) {j==1}
   return *p;
}
```

(b) Résultat sans $\mathcal{PT}^{\#}$

```c
//  T(main) {main==0}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i==0, i#init==1, j==1}
   *p = 0;

//  T(main) {i==0, j==1, main==0}
   return *p;
}
```

(c) Résultat avec $\mathcal{PT}^{\#}$

FIGURE E.2 – Code avec déréférencement

Le deuxième exemple *Fig.* E.3 permet de modifier la variable `i` en y accédant par différents déréférencements dont un double déréférencement. Ainsi, on fait pointer `p` sur `i`, `pp` sur `p`. Puis on dit que `q` pointe sur la même chose que `*pp` à savoir `i`. Enfin, on modifie `i` par nos différents pointeurs.

Les résultats présentés en E.3c montrent bien que `i` vaut successivement `0`, puis `1`, ensuite `2` et enfin `3`.

```c
int main() {
  int i = 0;
  int *p, *q, **pp;

  p = &i;
  pp = &p;
  //On veut avoir p=q
  q = *pp;

  //on modifie i
  *q = 1;
  *p=2;
  **pp = 3;

  return 0;
}
```

(a) Code à analyser

```c
//  T(main) {main==0}
int main() {
//  T(i) {i==0}
   int i = 0;
//  T() {i==0}
   int *p, *q, **pp;

//  T() {i==0}
   p = &i;
//  T() {i==0}
   pp = &p;
//  T() {i==0}
   //On veut avoir p=q
   q = *pp;

//  T(i) {i#init==0}
   //on modifie i
   *q = 1;
//  T(i) {}
   *p = 2;
//  T(i) {}
   **pp = 3;

//  T(main) {main==0}
   return 0;
}
```

(b) Résultat sans $\mathcal{PT}^{\#}$

```c
//  T(main) {main==0}
int main() {
//  T(i) {i==0}
   int i = 0;
//  T() {i==0}
   int *p, *q, **pp;

//  T() {i==0}
   p = &i;
//  T() {i==0}
   pp = &p;
//  T() {i==0}
   //On veut avoir p=q
   q = *pp;

//  T(i) {i==1, i#init==0}
   //on modifie i
   *q = 1;
//  T(i) {i==2, i#init==1}
   *p = 2;
//  T(i) {i==3, i#init==2}
   **pp = 3;

//  T(main) {i==3, main==0}
   return 0;
}
```

(c) Résultat avec $\mathcal{PT}^{\#}$

FIGURE E.3 – Code avec double déréférencement

Dans l'exemple *Fig.* E.4, on souhaite vérifier que l'on effectue bien une enveloppe convexe pour les affectations lorsqu'un pointeur peut pointer vers différentes valeurs. Cette enveloppe convexe est effectuée aussi bien avec un élément gauche qu'un élément droit d'une affectation. Ainsi, le code E.4a fait pointer `q` soit vers `m`, soit vers `n` et `p` soit vers `i`, soit vers `j`. Puis, `*p` est affecté à `*q`, c'est-à-dire que soit `i` vaut `m` ou `n` et `j` n'est pas modifié, soit `i` n'est pas modifié et `j` vaut `m` ou `n`.

Pour vérifier les résultats obtenus E.4c, on les compare aux résultats obtenus par un code témoin E.4e. Ce code témoin E.4d effectue les mêmes opérations décrites précédemment mais sans pointeurs. On peut constater que les résultats sont identiques. Ainsi, en supposant que l'analyse de pointeurs est correcte alors notre analyse avec les pointeurs est également correcte.

```c
#include<stdlib.h>

int main() {
  int i, j, m, n, *p, *q;
  i=0;
  j=1;
  m=10;
  n =11;

  if (rand()) {
    q = &m;
  } else {
    q = &n;
  }

  if (rand()) {
    p = &i;
  } else {
    p = &j;
  }

  *p = *q;
  return 0;
}
```

(a) Code à analyser

```c
// T(main) {main==0}
int main() {
// T(i,j,m,n) {}
   int i, j, m, n, *p, *q;
// T(n) {i==0, j==1, m==10, n==11}
  i=0; j=1; m=10; n =11;

// T() {i==0, j==1, m==10, n==11}
   if (rand())
// T() {i==0, j==1, m==10, n==11}
      q = &m;
   else
// T() {i==0, j==1, m==10, n==11}
      q = &n;
// T() {i==0, j==1, m==10, n==11}

   if (rand())
// T() {i==0, j==1, m==10, n==11}
      p = &i;
   else
// T() {i==0, j==1, m==10, n==11}
      p = &j;

// T(i,j) {i#init==0, j#init==1,
     m==10, n==11}
   *p = *q;
// T(main) {m==10, main==0, n==11}
   return 0;
}
```

(b) Résultat sans $\mathcal{PT}^{\#}$

```c
// T(main) {main==0}
int main() {
// T(i,j,m,n) {}
   int i, j, m, n, *p, *q;
// T(n) {i==0, j==1, m==10, n==11}
  i=0; j=1; m=10; n =11;

// T() {i==0, j==1, m==10, n==11}
   if (rand())
// T() {i==0, j==1, m==10, n==11}
      q = &m;
   else
// T() {i==0, j==1, m==10, n==11}
      q = &n;

// T() {i==0, j==1, m==10, n==11}
   if (rand())
// T() {i==0, j==1, m==10, n==11}
      p = &i;
   else
// T() {i==0, j==1, m==10, n==11}
      p = &j;

// T(i,j) {i#init==0, j#init==1,
     m==10, n==11, 0<=i, 100<=9i+10j,
      10i+11j<=121, 1<=j}
   *p = *q;

// T(main) {m==10, main==0, n==11,
     0<=i, 100<=9i+10j,
     10i+11j<=121, 1<=j}
   return 0;
}
```

(c) Résultat avec $\mathcal{PT}^{\#}$

```c
#include<stdlib.h>

int main() {
  int i, j, m, n;
  i=0;
  j=1;
  m=10;
  n =11;

  if (rand()) {
    i = rand()?m:n;
  } else {
    j = rand()?m:n;
  }

  return 0;
}
```

(d) Code témoin

```c
// T(main) {main==0}
int main() {
// T(i,j,m,n) {}
   int i, j, m, n;
// T(n) {i==0, j==1, m==10, n==11}
  i=0; j=1; m=10; n =11;

   int i, j, m, n;
// T(n) {i==0, j==1, m==10, n==11}
   n = 11;

// T(i,j) {i#init==0, j#init==1, m==10, n==11, 0<=i, 100<=9i+10j, 10i+11j
     <=121, 1<=j}
   if (rand())
// T(i) {i#init==0, j==1, m==10, n==11, 10<=i, i<=11}
      i = rand()?m:n;
   else
// T(j) {i==0, j#init==1, m==10, n==11, 10<=j, j<=11}
      j = rand()?m:n;

// T(main) {m==10, main==0, n==11, 0<=i, 100<=9i+10j, 10i+11j<=121, 1<=j}
   return 0;
}
```

(e) Résultat témoin

FIGURE E.4 – Code avec déréférencement et test

F Résultat de l'implémentation utilisant les chemins d'accès constant $\mathcal{CP}^{\#}$

L'utilisation des chemins d'accès constant $\mathcal{CP}^{\#}$ permet d'analyser les champs de structure ainsi que les valeurs d'une case d'un tableau. Certaines corrections restent tout de même à faire concernant ce dernier point.

Cette analyse nécessite la propriété supplémentaire *SEMANTICS_ANALYZE_CONSTANT_PATH*. Nos exemples utiliseront les scripts *tpips Fig.* F.1.

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE
setproperty SEMANTICS_ANALYZE_CONSTANT_PATH FALSE

activate PROPER_EFFECTS_WITH_POINTS_TO
activate TRANSFORMERS_INTER_FULL_WITH_POINTS_TO
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(a) Script sans $\mathcal{CP}^{\#}$

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE
setproperty SEMANTICS_ANALYZE_CONSTANT_PATH TRUE

activate PROPER_EFFECTS_WITH_POINTS_TO
activate TRANSFORMERS_INTER_FULL_WITH_POINTS_TO
activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(b) Script avec $\mathcal{CP}^{\#}$

FIGURE F.1 – Scripts *tpips* pour comparer l'analyse avec et sans $\mathcal{CP}^{\#}$

Le premier exemple F.2 effectue une analyse sur un champ d'une structure. On modifie ce champ de structure en y accédant de deux manières différentes à savoir soit directement avec `.first` soit en présence de pointeurs avec `->first`.

Les résultats F.2c permettent d'observer que le champ `first` de la variable `toto` vaut `0` puis `1`. Une amélioration quand au rendu visuel reste encore à faire, à savoir écrire `toto.first` au lieu de `toto[first]`.

```
struct Mastruct {
  int first;
  char second;
};

int main() {
  struct Mastruct toto;
  struct Mastruct *p;
  p = &toto;

  toto.first = 0;

  p->first = 1;

  return 0;
}
```

(a) Code à analyser

```
//  T(main) {main==0}
int main() {
//  T() {}
   struct Mastruct toto;
//  T() {}
   struct Mastruct *p;
//  T() {}
   p = &toto;

//  T() {}
   toto.first = 0;

//  T() {}
   p->first = 1;

//  T(main) {main==0}
   return 0;
}
```

(b) Résultat sans $\mathcal{CP}^{\#}$

```
//  T(main) {main==0}
int main() {
//  T() {}
   struct Mastruct toto;
//  T() {}
   struct Mastruct *p;
//  T() {}
   p = &toto;

//  T(toto[first]) {toto[first]==0}
   toto.first = 0;

//  T(toto[first]) {toto[first]==1,
      toto[first]#init==0}
   p->first = 1;

//  T(main) {main==0, toto[first]==1}
   return 0;
}
```

(c) Résultat avec $\mathcal{CP}^{\#}$

FIGURE F.2 – Code avec structure

Le deuxième exemple F.3 effectue une analyse sur les éléments d'un tableau initialisé manuellement, c'est-à-dire case par case. Ainsi, on donne la valeur `0` à la première case du tableau et `1` à la deuxième. On obtient bien les résultats attendus avec notre analyse F.3c.

```
  int a[2];

  a[0] = 0;
  a[1] = 1;

  return 0;
}
```

(a) Code à analyser

```
// T(main) {main==0}
int main() {
// T() {}
   int a[2];

// T() {}
   a[0] = 0;
// T() {}
   a[1] = 1;

// T(main) {main==0}
   return 0;
}
```

(b) Résultat sans $\mathcal{CP}^{\#}$

```
// T(main) {main==0}
int main() {
// T() {}
   int a[2];

// T(a[0]) {a[0]==0}
   a[0] = 0;
// T(a[1]) {a[0]==0, a[1]==1}
   a[1] = 1;

// T(main) {a[0]==0, a[1]==1,
   main==0}
   return 0;
}
```

(c) Résultat avec $\mathcal{CP}^{\#}$

FIGURE F.3 – Code avec tableau

Dans l'exemple F.4, on initialise un tableau automatiquement au moyen d'une boucle.

Les résultats obtenus F.4c pour le corps de boucle sont vrais mais ne sont pas réellement satisfaisants. En effet, `a[i]` ne fait pas partie des $\mathcal{CP}^{\#}$, donc `a[i]` ne devrait pas apparaitre dans nos *transfomers*, ainsi soit on devrait avoir `a[0]==0`, `a[1]==1`, ... au risque d'avoir une liste très longue, soit `0<=a[*]<=9`, soit n'avoir aucune information sur les cases de `a`. De plus le dernier *transfomer* est faux, `a[i]==9` avec `i==10` n'existe pas étant donné que notre tableau `a` n'a que dix cases.

Le choix le plus judicieux me semble lorsque l'on rencontre `a[i]` de le convertir en `a[*]` pour effectuer l'analyse. Il faut toutefois faire attention à bien considérer `a[*]` comme un ensemble de variables et non comme étant une variable précise. Ainsi, on ne doit pas affecter une valeur à `a[*]` mais la rajouter à `a[*]`, c'est-à-dire ne pas faire `a[*]==i` mais prendre l'enveloppe convexe des valeurs de `a[*]` et `i` pour définir les nouvelles valeurs de `a[*]`.

Avec cette méthodologie, `a[*]` représente uniquement l'ensemble des cases du tableau `a` qui ont été modifiées lorsque l'accès au tableau s'est fait au moyen d'une variable. Ainsi, si une case du tableau `a` n'a pas été modifiée, on considère que `a[*]` ne le contient pas. De même si la modification s'est faite sans variable, par exemple `a[0] = 0;`. Si l'on souhaite que le dernier cas soit également pris en compte pour l'ensemble `a[*]`, alors lorsque le cas se présente, il faut vérifier si l'ensemble `a[*]` existe, si oui rajouter la nouvelle valeur, si non le créer et rajouter la valeur.

```
int main() {
  int i, a[10];

  for(i=0; i<10; i++) {
    a[i] = i;
  }

  return 0;
}
```

(a) Code à analyser

```
// T(main) {main==0}
int main() {
// T(i) {}
   int i, a[10];

// T(i) {0<=i, i<=9}
   for(i = 0; i <= 9; i += 1)
// T() {0<=i, i<=9}
      a[i] = i;

// T(main) {i==10, main==0}
   return 0;
}
```

(b) Résultat sans $\mathcal{CP}^{\#}$

```
// T(main) {main==0}
int main() {
// T(i) {}
   int i, a[10];

// T(a[i],i) {0<=i, i<=9}
   for(i = 0; i <= 9; i += 1)
// T(a[i]) {a[i]==i, 0<=a[i], a[i]<=9}
      a[i] = i;

// T(main) {a[i]==9, i==10, main==0}
   return 0;
}
```

(c) Résultat avec $\mathcal{CP}^{\#}$

FIGURE F.4 – Code avec tableau et boucle

G Résultat de l'analyse sémantique des pointeurs

Cette analyse des pointeurs correspond à une analyse relationnelle. Ainsi, elle doit permettre de mettre en relation des informations sur les pointeurs eux-mêmes et non plus uniquement sur les valeurs pointées.

Cette analyse nécessite la propriété *SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES*. Nos exemples utiliseront donc les scripts *tpips* G.1. Sans cette propriété, les exemples présentés ne donnent aucune information. On compare notre analyse des pointeurs, avec l'analyse *points-to* déjà présente.

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty ABSTRACT_HEAP_LOCATIONS "context-sensitive"
setproperty ALIASING_ACROSS_TYPES FALSE

activate PRINT_CODE_POINTS_TO_LIST
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(a) Script pour l'analyse *points-to*

```
setenv WSPACE=nom_fichier
delete $WSPACE
create $WSPACE $WSPACE.c

setproperty ABORT_ON_USER_ERROR TRUE
setproperty SEMANTICS_COMPUTE_TRANSFORMERS_IN_CONTEXT TRUE
setproperty SEMANTICS_FIX_POINT_OPERATOR "derivative"
setproperty SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES TRUE

activate PRINT_CODE_TRANSFORMERS
display PRINTED_FILE[main]

close
delete $WSPACE
quit
```

(b) Script pour l'analyse sémantique des pointeurs

FIGURE G.1 – Scripts *tpips* pour l'analyse des pointeurs

Pour notre premier exemple G.2, on souhaite mettre en relation deux pointeurs `p` et `q`. Ainsi, on souhaite pouvoir dire que si `p` pointe vers `i` alors `q` pointe vers `j` et vice versa.

Les résultats de notre analyse G.2c permettent ainsi de donner la contrainte que `p+q` est égal à `&i+&j`. Donc `p` et `q` pointent soit sur `i` soit sur `j`, mais ils ne pointent tous les deux pas vers la même variable.

L'information *points-to* G.2b permet de dire que `p` pointe soit sur `i`, soit sur `j`, de même pour `q`. Mais elle n'indique pas que `p` et `q` pointent tous les deux sur des variables différentes.

Notre deuxième exemple G.3 présente de l'arithmétique sur pointeurs. Ainsi, à partir d'un pointeur `q` qui pointe vers le premier élément d'un tableau `&a[0]`, on souhaite définir un second pointeur `p` qui pointera vers un autre élément du tableau, dans notre cas la sixième case.

Pour rappel, le choix fait pour traiter l'arithmétique sur les pointeurs est d'ajouter la taille du type du pointeur étudié *Sec.* 4.5.

Les résultats de notre analyse G.3c sont bien que `p` part de `&a[0]` auquel on ajoute six fois la taille du type du tableau. L'autre solution présentée en *Sec.* 4.5 devrait donner le résultat suivant `p==&a[6]`.

Les résultats fournis par l'analyse *points-to* G.3b permettent juste de dire que `p` pointe vers un élément du tableau `a` sans avoir aucune idée de cet élément.

Le troisième exemple G.4 effectue un test entre deux pointeurs `p` et `q`. Dans cet exemple, on définit que `q=p`. Ainsi, lorsque l'on effectue le test, on ne doit pas rentrer dans le cas `p!=q`, et `i` vaudra `2` à la fin.

Les résultats de notre analyse G.4c indiquent bien que le cas de test où `p!=q` est inatteignable avec la présence du transformer `0==-1`. Alors que l'analyse *points-to* G.4b ne l'a pas

```c
#include <stdlib.h>

int main() {
  int i, j, *p, *q;

  if (rand()) {
    p = &i;
    q = &j;
  } else {
    p = &j;
    q = &i;
  }

  return 0;
}
```

(a) Code à analyser

```c
int main() {
// Points To: none
   int i, j, *p, *q;

// p->undefined, EXACT
// q->undefined, EXACT
   if (rand()) {
// p->undefined, EXACT
// q->undefined, EXACT
      p = &i;
// p->i, EXACT
// q->undefined, EXACT
      q = &j;
   }
   else {
// p->undefined, EXACT
// q->undefined, EXACT
      p = &j;
// p->j, EXACT
// q->undefined, EXACT
      q = &i;
   }

// p->i, MAY; p->j, MAY
// q->i, MAY; q->j, MAY
   return 0;
}
```

(b) Résultat analyse *points-to*

```c
//  T(main) {main==0}
int main() {
//  T(i,j,p,q) {}
   int i, j, *p, *q;

//  T(p,q) {&i+&j==p+q}
   if (rand()) {
//  T(p) {&i==p}
      p = &i;
//  T(q) {&i==p, &j==q}
      q = &j;
   }
   else {
//  T(p) {&j==p}
      p = &j;
//  T(q) {&i==q, &j==p}
      q = &i;
   }

//  T(main) {&i+&j==p+q, main==0}
   return 0;
}
```

(c) Résultat analyse des pointeurs

FIGURE G.2 – Code avec référencement et test

```c
#include <stdlib.h>

int main() {
  int a[20], i=2, j=3;
  int *p, *q;

  q=&a[0];
  p=q+i*j;

  return 0;
}
```

(a) Code à analyser

```c
int main() {
// Points To: none
   int a[20], i = 2, j = 3;
// Points To: none
   int *p, *q;

// p->undefined, EXACT
// q->undefined, EXACT
   q = &a[0];
// p->undefined, EXACT
// q->a[0], EXACT
   p = q+i*j;

// p->a[*], MAY
// q->a[0], EXACT
   return 0;
}
```

(b) Résultat analyse *points-to*

```c
//  T(main) {main==0}
int main() {
//  T(i,j) {i==2, j==3}
   int a[20], i = 2, j = 3;
//  T(p,q) {i==2, j==3}
   int *p, *q;

//  T(q) {&a[0]==q, i==2, j==3}
   q = &a[0];
//  T(p) {&a[0]+6sizeof(int)==p,
      &a[0]==q, i==2, j==3}
   p = q+i*j;

//  T(main) {&a[0]+6sizeof(int)==p,
      &a[0]==q, i==2, j==3, main==0}
   return 0;
}
```

(c) Résultat analyse des pointeurs

FIGURE G.3 – Code avec arithmétique sur pointeur détecté. L'analyse *points-to* indique une zone de code qui n'est pas atteint avec l'ensemble *unreachable*.

```
int foo(int *p) {
  int *q = p;
  int i=0;

  if(q!=p)
    i = 1;
  else
    i = 2;

  return i;
}
```

(a) Code à analyser

```
int foo(int *p) {
// p->*NULL*, MAY ; p->_p_1[0], MAY
   int *q = p;

// p->*NULL*, MAY ; p->_p_1[0], MAY
// q->*NULL*, MAY ; q->_p_1[0], MAY
   int i = 0;

// p->*NULL*, MAY ; p->_p_1[0], MAY
// q->*NULL*, MAY ; q->_p_1[0], MAY
   if (q!=p)
// p->*NULL*, MAY; p->_p_1[0], MAY
// q->*NULL*, MAY; q->_p_1[0], MAY
      i = 1;
   else
// p->*NULL*, MAY ; p->_p_1[0], MAY
// q->*NULL*, MAY ; q->_p_1[0], MAY
      i = 2;

// p->*NULL*, MAY ; p->_p_1[0], MAY
// q->*NULL*, MAY ; q->_p_1[0], MAY
   return i;
}
```

(b) Résultat analyse *points-to*

```
//  T(foo) {foo==2}
int foo(int *p) {
//  T(q) {p==q}
   int *q = p;
//  T(i) {i==0, p==q}
   int i = 0;

//  T(i) {i==2, i#init==0, p==q}
   if (q!=p)
//  T() {0==-1}
      i = 1;
   else
//  T(i) {i==2, i#init==0, p==q}
      i = 2;

//  T(foo) {foo==2, i==2, p==q}
   return i;
}
```

(c) Résultat analyse des pointeurs

FIGURE G.4 – Code avec test entre pointeurs

Notre dernier exemple G.5 présente les limites de notre analyse des pointeurs et une amélioration possible de celle-ci. Dans cet exemple, on initialise aléatoirement les variables vers lesquelles pointent `p` et `q`. On sait néanmoins qu'ils ne pointeront jamais vers les mêmes variables. Ainsi, lorsque l'on arrive au test, le premier cas `p==q` ne devrait pas arriver.

L'analyse *points-to* G.5b détecte bien que le premier cas du test n'arrive jamais. Ceci est indiqué avec l'ensemble *unreachable*.

Notre analyse G.5c ne permet pas de le détecter. La raison est que notre analyse est une analyse relationnelle. Or lors de l'initialisation aléatoire de `p` et `q`, il n'y a pas de relation ni entre `p` et `q`, ni entre les adresses dont ils prennent la valeur. En effet, on ne peut pas établir de relation entre des adresses[12]. Ainsi, lorsque l'on teste `p==q`, on ne peut rien dire sur le résultat de ce dernier et tous les cas doivent être faits.

Une amélioration de notre analyse des pointeurs pourrait être apportée en y ajoutant la possibilité d'utilisée le graphe $\mathcal{PT}^{\#}$ lors d'un test sur les pointeurs. Ainsi, lorsque l'on arrive à un test entre pointeur, on essaye dans un premier temps de récupérer des informations sans le graphe $\mathcal{PT}^{\#}$, cela permet de garder la résolution de notre exemple G.4. Puis si cela n'a rien apporté, et si les pointeurs à comparer pointent vers des variables différentes de *undefined*, on effectue la comparaison sur les variables pointées.

12. Une exception peut être faite entre les adresses des cases d'un même tableau.

```c
#include<stdlib.h>

int main() {
  int *p, *q;
  int i, j, k, l, r=0;

  if(rand()) {
    p = &i;
    q = &j;
  }
  else {
    p = &k;
    q = &l;
  }

  if(p==q)
    r = 2;
  else
    r = 1;

  return r;
}
```

(a) Code à analyser

```c
int main() {
// Points To: none
   int *p, *q;
   int i, j, k, l, r = 0;

// p->undefined, EXACT
// q->undefined, EXACT
   if (rand()) {
      p = &i;
      q = &j;
   }
   else {
      p = &k;
      q = &l;
   }

// p->i, MAY ; p->k, MAY
// q->j, MAY ; q->l, MAY
   if (p==q)
// Points To: unreachable
      r = 2;
   else
// p->i, MAY ; p->k, MAY
// q->j, MAY ; q->l, MAY
      r = 1;

// p->i, MAY ; p->k, MAY
// q->j, MAY ; q->l, MAY
   return r;
}
```

(b) Résultat analyse *points-to*

```c
//  T(main) {1<=main, main<=2}
int main() {
//  T(p,q) {}
   int *p, *q;
//  T(c,i,j,k,l,r) {r==0}
   int i, j, k, l, r = 0;

//  T(p,q) {r==0}
   if (rand()) {
//  T(p) {&i==p, r==0}
      p = &i;
//  T(q) {&i==p, &j==q, r==0}
      q = &j;
   }
   else {
//  T(p) {&k==p, c==0, r==0}
      p = &k;
//  T(q) {&k==p, &l==q, c==0, r==0}
      q = &l;
   }

//  T(r) {r#init==0, 1<=r, r<=2}
   if (p==q)
//  T(r) {p==q, r==2, r#init==0}
      r = 2;
   else
//  T(r) {r==1, r#init==0}
      r = 1;

//  T(main) {main==r, 1<=main,
      main<=2}
   return r;
}
```

(c) Résultat analyse des pointeurs

FIGURE G.5 – Code avec test entre pointeurs 2

Table des matières

Table des figures

Références

[ACI10] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1) :3–16, October 2010.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

[Ast] Astrée. Astrée : Analyseur statique de logiciels temps-réel embarqués. site officiel : http://www.astree.ens.fr/.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CET] CETUS. Cetus : A source-to-source compiler infrastructure for c programs. site officiel : http://cetus.ecn.purdue.edu/.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Cre96] Béatrice Creusillet. *Array Region Analyses and Applications*. PhD thesis, École nationale supérieure des mines de Paris, 1996. http://www.cri.ensmp.fr/classement/doc/A-295.pdf.

[Gor79] Michael J. C. Gordon. *The denotational description of programming languges*. Springer-Verlag, 1979.

[Hin01] Michael Hind. Pointer analysis : haven't we solved this problem yet ? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use ? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[ISO07] norme C99 International Standardization Organization. ISO/IEC 9899:TC3. Technical report, 2007.

[Loo] LooPo. LooPo : Polyhedral loop parallelization. site officiel : http://www.infosun.fim.uni-passau.de/cl/loopo/.

[LR04] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39(4) :473–489, April 2004.

[Men13] Amira Mensi. *Analyse des pointeurs pour le langage C*. PhD thesis, École nationale supérieure des mines de Paris, 2013.

[Min06] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. *SIGPLAN Not.*, 41(7) :54–63, June 2006. http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf.

[Min13] Antoine Miné. MPRI, cours3 : Relational numerical abstract domains, 2012-2013.

[OSC] OSCAR. OSCAR : Optimally scheduled advanced multiprocessor. site officiel : http://www.kasahara.elec.waseda.ac.jp/intro.en.html.

[PIP] PIPS. PIPS : Automatic parallelizer and code transformation framework. Le site officiel de PIPS : http://www.pips4u.org/.

[PoC] PoCC. PoCC : the polyhedral compiler collection. site officiel : http://www.cse.ohio-state.edu/~pouchet//software/pocc/doc/htmldoc/htmldoc/main.html.

[ROS] ROSE. ROSE@LLNL making compiler technology accessible. site officiel : http://rosecompiler.org/.